A Fast Simulator and Debugger for a Network Processor

Jens Wagner
University of Dortmund
Computer Science XII
Embedded Systems Group
email: Jens.Wagner@uni-dortmund.de

Rainer Leupers
Aachen University of Technology
Dept. of EE & IT
Integrated Signal Processing Systems
email: leupers@icd.de

1 Introduction

Due to short time-to-market requirements and the need for flexibility, many of today's embedded system designs are based on software executed on programmable embedded processors. While custom hardware (ASICs) offers the highest computational efficiency (in MIPS/Watt), the growing complexity of embedded systems makes the use of programmable building blocks mandatory. However, the efficiency gap between ASICs and standard "off-the-shelf" processors may be several orders of magnitude, which is mostly not acceptable in embedded system design.

Therefore, *application-specific instruction set processors* (ASIPs) are being used as a compromise between ASICs and general purpose processors. ASIPs are programmable, but their architecture is tuned for a certain application domain, e.g., audio or video processing. In this paper, we focus on an ASIP designed for *network processing*.

While ASIPs form a good comprise from a hardware efficiency perspective, a major problem in ASIP-based system design is the need for software development tool support, including compilers, assemblers, simulators, and debuggers [1]. Standard processors mostly come with a corresponding software development tool chain, but for each new ASIP such tools need to be developed from scratch. In addition, there is a need for high tool performance, since a large amount of input data processing has to be simulated in order to validate a design. In particular, this holds for compute-intensive domains like signal or network processing.

In this paper, we describe the design of an efficient simulator/debugger tool environment for an industrial network processor. The tools offer two main advantages:

- The simulator is based on the *compiled simulation* principle. Instead of the classical interpretive simulation approach, it simulates processor instructions by means of compiled C++ code on a simulation host. This makes the simulator highly efficient [2].

- The debugger, which builds on the compiled simulator, has been linked to the popular software debugger DDD from TU Braunschweig [3]. DDD is a comfortable graphical tool, that supports all common debugger functionality like breakpoints, single-stepping, and status monitoring. By relying on DDD, we were able to reuse an existing, stable debugger kernel and GUI, while only the processor-specific simulation routines had to be newly developed.

2 Target architecture

Our target architecture is an Infineon Technologies Network Processor (NP) [4]. The NP instruction set is optimized for efficient *protocol processing*. This is motivated by the growing need for new high bandwidth communication equipment in networks (e.g. Internet routers and Ethernet adapters) as well as in telecommunication (e.g. ISDN and xDSL). The corresponding communication protocols mostly employ bit stream oriented data formats. The bit streams consist of *packets* of different length, i.e. there are variable length header packets and (typically longer) payload packets. Typical packet processing requirements include decoding, compression, encryption, or routing.

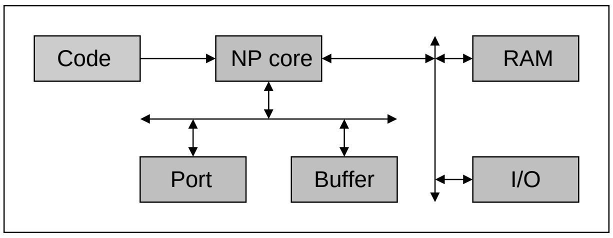


Figure 1. *Infineon NP architecture*

Fig. 1 shows the overall architecture. The NP core shows a RISC-like basic architecture with general-purpose registers and special extensions for bit-level data access, data and control ports, interrupts, and fast context switching.

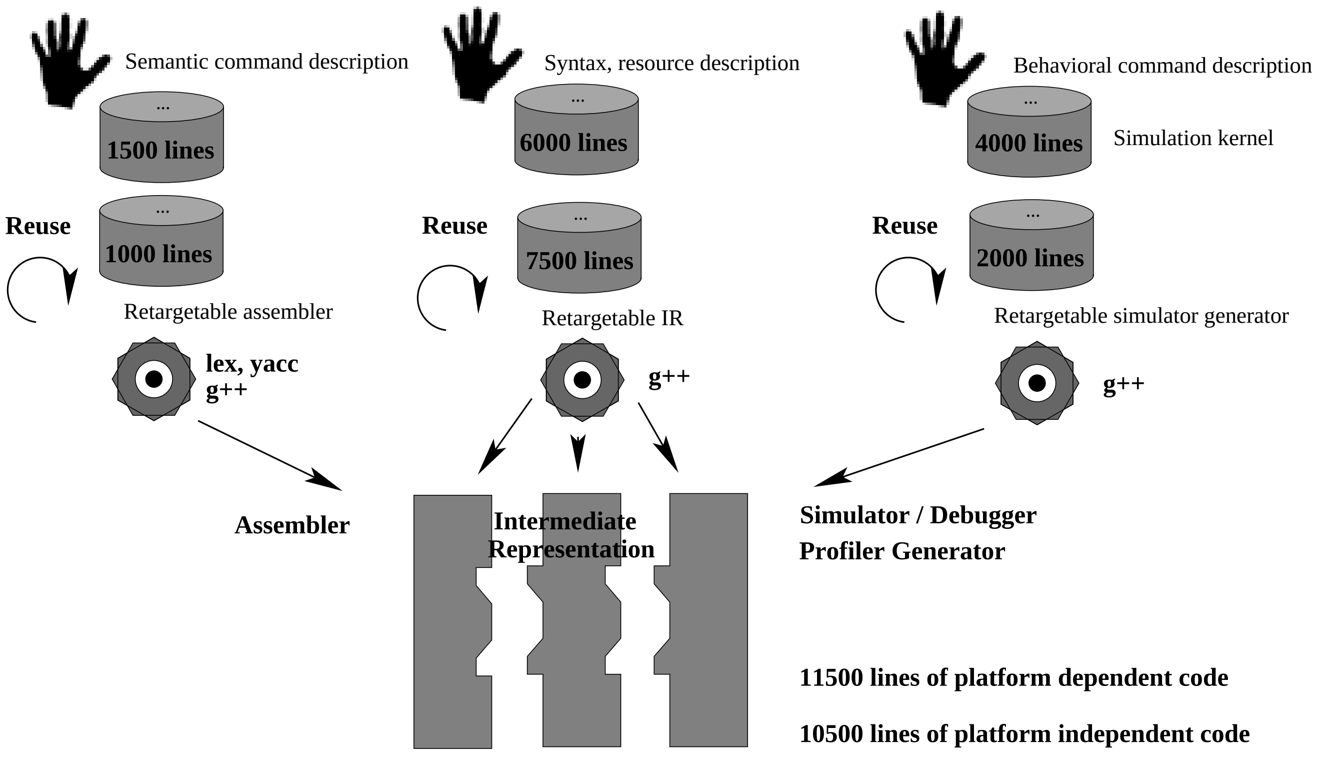
3 Simulation technique

3.1 Tool construction

In order to reduce development costs and to accelerate the tool development process our tools are systematically constructed on a framework of reusable libraries. These libraries are proven in many projects and retained unchanged for the retargeting of the tool suite to a new target processor. Parts of these tools are not only used for assemblers/simulators, but also for retargetable ANSI C compilers [5]. Our tools are built by means of a completely object oriented design, except for parser and scanner which are constructed from a lex/yacc grammar description. This implies a high percentage of reusable code.

Figure 2 shows the amount of retargetability of the tool chain. A rate of 47 % of platform independent code is shown as worst case. The rate is much higher if not all of the hardware dependent parts of the sources have to be written from scratch. Usually, even for different processors much of the hardware dependent implementation can be reused.

The hardware dependent part of the toolchain is isolated in three libraries, as shown in fig. 2. First the semantic description of the assembly commands (e.g. the meaning of assembly commands like *.org* or *label*). The second library contains all information about assembly syntax and processor specific



Semantic command description
1500 lines
1000 lines
Reuse
Retargetable assembler
lex, yacc
g++
Assembler
Syntax, resource description
6000 lines
7500 lines
Reuse
Retargetable IR
g++
Intermediate Representation
Behavioral command description
4000 lines
Simulation kernel
2000 lines
Reuse
Retargetable simulator generator
g++
Simulator / Debugger
Profiler Generator
11500 lines of platform dependent code
10500 lines of platform independent code

Figure 2. *Retargebility of the simulator source code*

resource names (e.g. register names). The third library describes the behavior of all assembly commands (how an operation manipulates the processor state and which resources it occupies).

3.2 Simulator generation

The simulator is a stand-alone binary program. It contains all information about the target architecture and the application code. To create a new simulation the following steps are performed by the toolset. The simulator creation starts from an assembly source file or a executable binary file. The assembly file would be assembled (and linked) for resource binding, the executable file is disassembled to generate readable source code information.

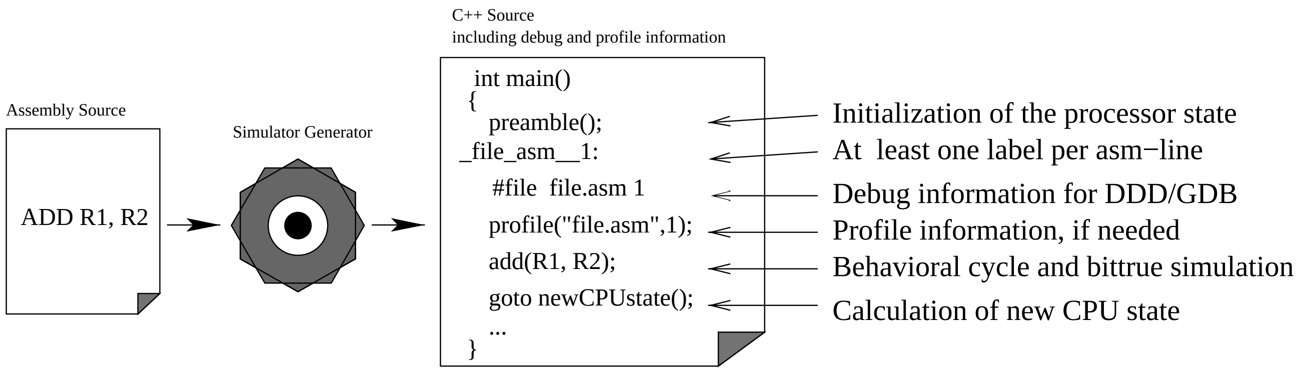
The Simulator Generator creates the source code of the simulator from the input data, the assembly source file and the assembled binary. The simulator source is compiled with the GNU C++ Compiler and linked with a library containing the behavioral description of the target processor. The simulator generator follows three different strategies, dependent on user settings:

Stand alone simulator: The created simulator is as fast as possible. The Simulator Generator makes extensive use of function inlining to enhance the simulation speed. The resulting simulator is still cycle and bit true, but temporary results of operations may invisible because of optimizations.

Profiler: Additional code is inserted for statistical investigation. The resulting simulator is cycle and bit true and still speed optimized but significantly slower than the stand alone simulator.

Debugger: The simulator generator inserts additional code for visualization of the CPU state, manipulation of the CPU state and also for support of execution in a debugging environment (e.g. breakpoints, stepping, run over function calls). The resulting code may one to two orders of magnitude slower than the stand alone simulator. The debug strategy can be combined with profiling.

Every time the user of the tool set changes the assembly input files the Simulator Generator creates a completely new simulator.



Assembly Source
ADD R1, R2
Simulator Generator
C++ Source
including debug and profile information
int main()
{
preamble();
_file_asm__1:
#file file.asm 1
profile("file.asm",1);
add(R1, R2);
goto newCPUstate();
...
}
Initialization of the processor state
At least one label per asm−line
Debug information for DDD/GDB
Profile information, if needed
Behavioral cycle and bittrue simulation
Calculation of new CPU state

Figure 3. *Transformation from assembly source to compilable C++ code*

Figure 3 sketches the transformation of an input assembly file. Special features like a program preamble for simulating a hardware reset are function calls to library functions. The complete assembly source is transformed into the main function of the simulator. Even if the assembly code is structured into functions the Simulator Generator models the structure of the assembly program with *goto* and *labels*. Every new assembly command forces a new label in the simulator source code. In a internal hash table the simulator stores the mapping from target processor addresses to simulator labels. Every manipulation of the Program Counter in the simulator which is not an increment by one will result in a jump to a label. The label is calculated by use of the hash table. Therefore the simulator can simulate every manipulation of the Program Counter even if it is not known at compile time. Debug information is made explicit by the ANSI C preprocessor *#file* directive. With the debug information we force a debugger to hide the source of the simulator from the user and to show the original assembly code instead. The profile information is created by inline function calls to library functions. The simulation of the behavior of the assembly command itself is done by a library call. The simulation code is linked or inlined from the behavioral library of the tool set. At the end of the simulation of an assembly command the new CPU state is set by a special function. This may be e.g. pipeline clock tick, manipulation of the Program Counter or the occurrence of an interrupt. The function *newCPUstate()* returns a pointer to a label of the next simulated assembly command in the address space of the simulator. This process is iterated for each assembly command in the assembly source file.

3.3 Simulation process

The classical approach to processor instruction set simulation is *interpretive*: Each instruction to be simulated is decoded from the binary program and is passed to an appropriate simulation routine, which interprets the instruction and correspondingly updates the simulator status. However, the interpretive technique is quite slow. In particular, instructions within loops cause a large overhead, since the decoding step is unnecessarily repeated over and over again. Typically, the interpretive approach achieves a simulation speed in the order of a few kilo-instructions per simulation host CPU second, which is insufficient for compute-intensive programs.

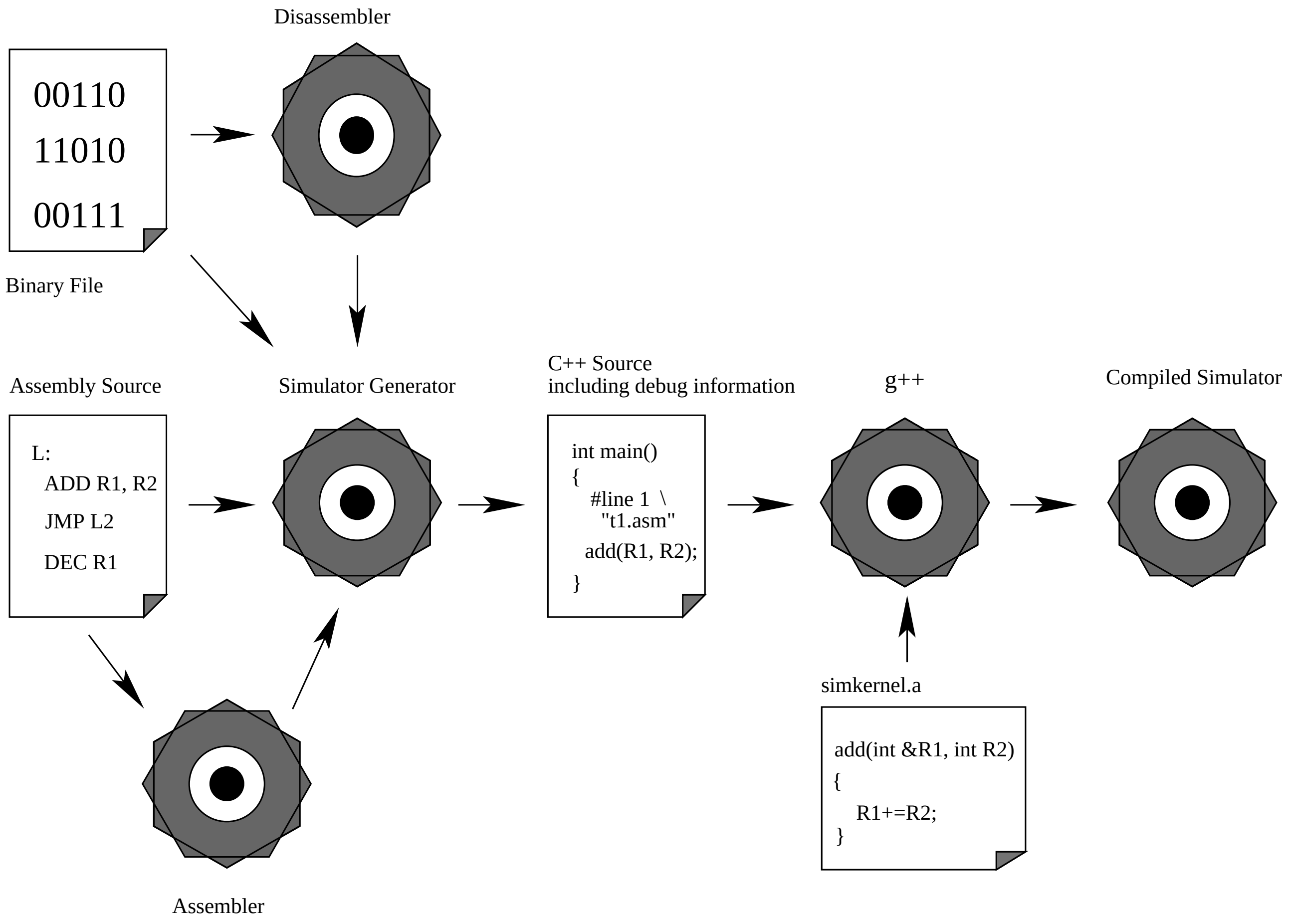


Figure 4. *Compiled simulation method*

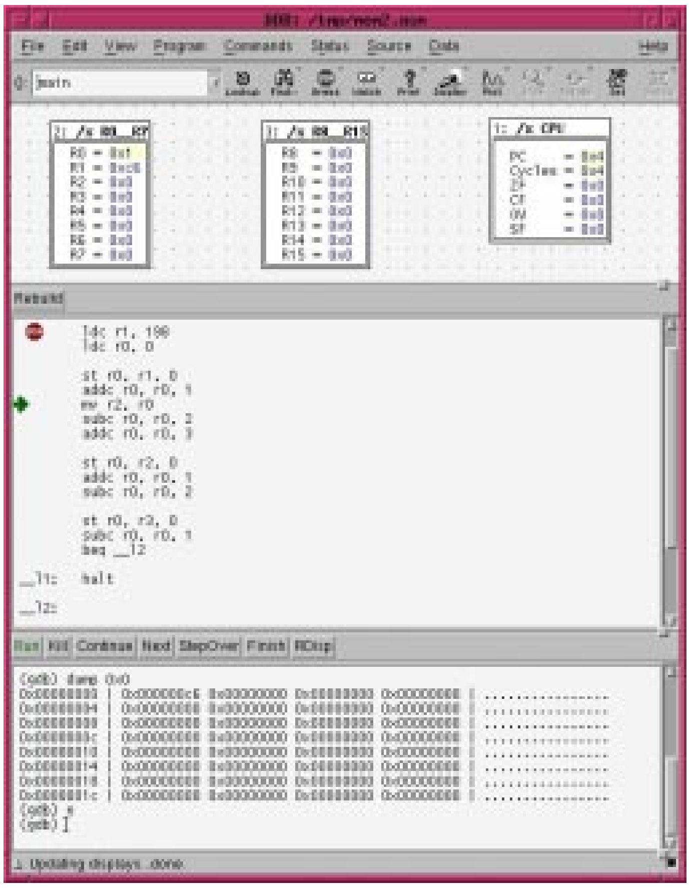
In contrast, in the *compiled* simulation approach (fig. 4), the simulator is not only processor-specific, but also *program-specific*: A C++ simulation program is generated for each binary application program. This results in a twofold performance gain: The instruction decoding overhead is eliminated, since the decoding is performed only once at the time of simulation program generation. In addition, the full optimization potential of the host C++ compiler can be exploited for fast simulator generation. As a result, our simulator achieves a speed of more than 500 kilo-instructions per CPU second. The simulation is cycle-true and bit-true. Generation of the simulation program itself takes only a few CPU seconds. The main restriction of compiled simulation is that self-modifying code is not supported. However, in the area of embedded systems, this is usually not a real drawback.

As shown in fig. 4 the *simulation generator* transforms an assembly source into a compilable C++ source code. This is essentially only a syntactic substitution. If a binary should be simulated it is deassembled first into an assembly source code. A disassembler can easily be generated from the hardware description libraries which are already available.

4 Graphical debugger

A very useful by-product of compiled simulation is that application programs can be debugged with existing tools. This is due to the fact that we use a C++ simulation program which in turn is compiled to an executable program on the simulation host, e.g. a Linux workstation. Hence, source-level debugging of the simulation program is possible.

The only problem is that the user normally does not want to see the tool-specific C++ simulation code in the debugger, but the assembly code of the target machine. However, this can be solved via bypassing the C++ simulation code and displaying assembly code in the debugger source window instead. The debugger interface keeps track of the correspondence between assembly line numbers and line numbers in the C++ simulation code. Therefore, the user does not need to know about the underlying simulation technique, but breakpoint setting, single-stepping, and other common debugger functions can be performed directly in the assembly code.



```
1: /x R0..R7        1: /x R8..R15        1: /x CPU
R0 = 0x1            R8  = 0x0            PC     = 0x4
R1 = 0xc6           R9  = 0x0            Cycles = 0x4
R2 = 0x0            R10 = 0x0            ZF     = 0x0
R3 = 0x0            R11 = 0x0            CF     = 0x0
R4 = 0x0            R12 = 0x0            OV     = 0x0
R5 = 0x0            R13 = 0x0            SF     = 0x0
R6 = 0x0            R14 = 0x0
R7 = 0x0            R15 = 0x0
```

```
        ldc r1, 198
        ldc r0, 0

        st r0, r1, 0
        addc r0, r0, 1
        mv r2, r0
        subc r0, r0, 2
        addc r0, r0, 3

        st r0, r2, 0
        addc r0, r0, 1
        subc r0, r0, 2

        st r0, r3, 0
        subc r0, r0, 1
        beq __l2

__l1:   halt

__l2:
```

```
(gdb) dump 0x0
0x00000000 | 0x000000c6 0x00000000 0x00000000 0x00000000 | ................
0x00000004 | 0x00000000 0x00000000 0x00000000 0x00000000 | ................
0x00000008 | 0x00000000 0x00000000 0x00000000 0x00000000 | ................
0x0000000c | 0x00000000 0x00000000 0x00000000 0x00000000 | ................
0x00000010 | 0x00000000 0x00000000 0x00000000 0x00000000 | ................
0x00000014 | 0x00000000 0x00000000 0x00000000 0x00000000 | ................
0x00000018 | 0x00000000 0x00000000 0x00000000 0x00000000 | ................
0x0000001c | 0x00000000 0x00000000 0x00000000 0x00000000 | ................
(gdb) s
(gdb) 
```

Figure 5. *DDD-based Network Processor debugger GUI*

We exploited these features of compiled simulation by connecting our NP processor simulator to the *data display debugger* (DDD) from TU Braunschweig. DDD is a graphical frontend for the GNU debugger GDB. It shows a comfortable GUI (fig. 5) and has a special window for graphical display of data structure contents. A custom configuration file developed for the NP processor ensures that upon invocation of DDD all relevant status information (e.g. register contents) is automatically displayed.

The use of DDD resulted in significant development time savings. Moreover, in contrast to classical interpretive simulators, there was no need to develop different tools for interactive debugging and fast stand-alone simulation. The underlying GDB debugger supports also the execution of target code invoked by debugger commands. We created a number of debugger commands using this feature. Figure 5 shows in the lower window a memory dump. For this feature we defined a user command in the DDD configuration. This user command calls a function in the simulator which dumps a part of the simulator memory to the standard output. The memory simulation is a framework of C++ classes in a behavioral library. It would be nearly impossible to investigate the memory content without a comfortable debugger.

5 Conclusions and future work

Modern embedded systems are frequently designed on the basis of programmable ASIPs, which allow for high flexibility and IP reuse. However, effective tool support for ASIP software is still frequently missing. In this contribution we outlined that it is possible to use fast compiled instruction set simulation in a comfortable debugging environment. Furthermore we showed that it is possible to clearly separate hardware dependent and independent code even in a simulator tool, which is traditionally a very processor-specific area. This allows for a high percentage of code reuse for the simulator tool set and hence accelerates simulator development and retargeting.

Acknowledgments

The simulator described in this paper has been developed at University of Dortmund and the Informatik Centrum Dortmund (ICD) for Infineon Technologies AG (Munich), whose project funding is gratefully acknowledged. The authors would also like to thank Jörg Eckart and Luis Gomez, who significantly contributed to the tool implementation and test.

References

[1] V. Zivojnovic: *DSP Processor / Compiler Co-Design: A Quantitative Approach*, Shaker Publishing, Aachen, 1998

[2] A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen A. Wieferink, H. Meyr: *A Novel Methology for the Design of Application-Specific Instruction-Set Processors (ASIPs) Using a Machine Description Language*, IEEE Transactions on Computer-Aided Design (TCAD), volume 20, number 11, 2001

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

[4] 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

[5] J. Wagner, R. Leupers: *C Compiler Design for a Network Processor*, IEEE Transactions on Computer-Aided Design (TCAD), volume 20, number 11, 2001