BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PROGRAMMING OF RECONFIGURABLE SYSTEMS USING A HIGHER PROGRAMMING LANGUAGE

PROGRAMOVÁNÍ REKONFIGUROVATELNÝCH SYSTÉMŮ POMOCÍ VYŠŠÍHO PROGRAMOVACÍHO JAZYKA

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR AUTOR PRÁCE	Ing. ADAM HUSÁR
SUPERVISOR VEDOUCÍ PRÁCE	prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2014

Abstract

The dissertation thesis is focused on application specific instruction set processors (ASIP) programming and optimization. The goal was to create an environment for fast ASIP optimization, which includes a higher level language compiler, and a processor template. Process of compiler generation is divided into two steps: semantics extraction and compiler backend generation. The author also designed several processor cores, one of them, a 32-bit extensible processor core Codix RISC, is described here. Together with the generated compiler, and the extensible processor core, the author created a tool for fast ASIP optimization.

Key Words

Compilers, processor architectures, application-specific instruction set processors, instruction set extensions, architecture description languages, embedded systems.

Contents

1 Introduction **2**
1.1 Thesis Goal Statement 3

2 State of The Art **6**
2.1 Application Specific Instruction Set Processor Design Tools 6
2.2 CodAL Language 6
2.3 High Level Language Compilers 8
2.3.1 Higher Language Level Compiler Structure 8
2.3.2 Retargetable Compilers 8
2.4 State of the Art - Conclusion 9

3 Solution **11**
3.1 Instruction Semantics Extraction 12
3.1.1 Semantics Extraction Results 14
3.2 Retargetable LLVM Compiler Generation 16
3.2.1 Overal Compiler Generator Design 16
3.2.2 Generated Compiler Results 17
3.3 Reconfigurable Processor Core Codix RISC 22
3.4 Manual ISE identification 25

4 Conclusion **28**
Bibliography 29
Curriculum Vitae 33

Chapter 1

Introduction

The theme of the thesis is reconfigurable computing systems, how to compile for them, and how to optimize them.

There are three usual sources of parallelism that can be exploited when speeding up program execution on a system with processors. The first one is the Thread Level Parallelism (TLP) in its various forms, which allows using multiple processor cores simultaneously. The second one is the Instruction Level Parallelism (ILP), where either the microarchitecture or the compiler finds instructions from the ISA that can be executed in parallel on the same processor core. The third source is the Data Level Parallelism (DLP), which allows computing multiple computations simultaneously by just one instruction.

A lot of research has been done in automatic parallelization (exploiting the TLP), but there will still be a part of an application that is inherently sequential. This sequential part then limits the maximal achievable speed-up as described by Amdahl's law. For example, even if we parallelize 80% of an application, so this part is computed instantly, the maximal speed-up is 500%, no matter how many more processors we put into the system. This is where exploitation of ILP, DLP, and speeding up the elementary arithmetic operations come into play.

Especially in signal and image processing algorithms, application-specific instruction set processors (ASIPs) are used with success for speeding up the sequential parts of an application. Application-specific instruction set processors are usually single-issue or very long instruction word (VLIW) processors extended with a combination of special instructions, special registers, look-up tables, local memories, and interfaces such as queues.

Lower power consumption and lower area predetermine ASIPs to be used in areas such as mobile handsets, wired and wireless networking, printers, home entertainment, performance-demanding peripheral controllers, and others.

In ASIPs, parallelism between elementary operations (ILP and DLP) is exploited statically by the use of wide instruction set extensions (ISEs), which compute several elementary operations in parallel. The elementary operations can then be accelerated using shorter datapaths between the functional units that execute them. In an ISE hardware implementation, functional units such as adders, multipliers, etc. are connected directly. No additional latency is added by passing intermediate results through forwarding paths or through registers, as is usual in standard processors.

Functional units can be simplified to a required bitwidth. For example, instead of a general 32-bit adder an 8-bit adder can be used where sufficient,. Finally, bit manipulation operations such as shifts, masks, bit swaps, zero or sign extends, etc. can be performed in hardware with minimal latency.

To sum up, faster computation of sequential calculations is enabled by: shorter datapaths between functional units, functional units with minimal required bitwidths, and bit manipulation operations. Wide ISEs allow exploiting elementary-operation level parallelism (ILP and DLP). The code size is also smaller since one ISE may replace tens of instructions. The new ISEs make the so-called semantic gap between the problem and the electronic components narrower, and allow a more efficient compilation from the problem domain to the electronic components.

The problem is now how to efficiently design such ASIPs with ISEs.

1.1 Thesis Goal Statement

Many tools are needed to compile programs for a processor, to simulate it, and to be able to quickly test and evaluate new extensions to a processor. All this can be done by manually changing the compiler, assembler, simulator, and other tools each time a new instruction is added. However, such approach is very time consuming, and tools are needed that automate this process. For this purpose, the Lissom project [3] was started at the Faculty of Information Technologies at Brno University of Technology in the year 2004. The goal of this project is to develop an ASIP design tool usable in practice. The tool is based on the ISAC architecture description language (ADL), which has been later substantially changed and renamed to CodAL.

The author, partially inspired by the success of the Tensilica Xtensa extensible processor core [14], and by suggestions from people knowledgeable about the research area, also added a new partial goal. Namely to provide an extensible processor core with a set of ASIP design tools. The user of the Lissom tools will not have to start their processor design from scratch. With an extensible processor core usable as a template, a new ASIP design can be finished much faster.

The design and optimization process proposed by the author is shown in figure 1.1. The process consists of the following steps: 1) take a template in the form of an ADL model, 2) generate the compiler, assembler, and simulator, 3) compile and profile the application, 4) using a profile and an intermediate program representation identify, either manually or automatically, new ISEs and add them to the ADL model, and use them also in the application's source code if necessary. Older ISEs can be removed from the model and the application. Steps 2)-4) are repeated until the performance and cost requirements are met.

The author started his doctoral studies in the year 2007. At that time, we already had an ADL language called ISAC, its parser, simulator generator, linker, and also assembler generator. The assembler generator was implemented by the author as his master's thesis [9].

Components that were missing at that time are shown in figure 1.1 as boxes with white background. The initial processor model template, compiler generator, other C compiler components, and support for either manual or automatic ISE identification and generation were missing. So the author started filling them in.

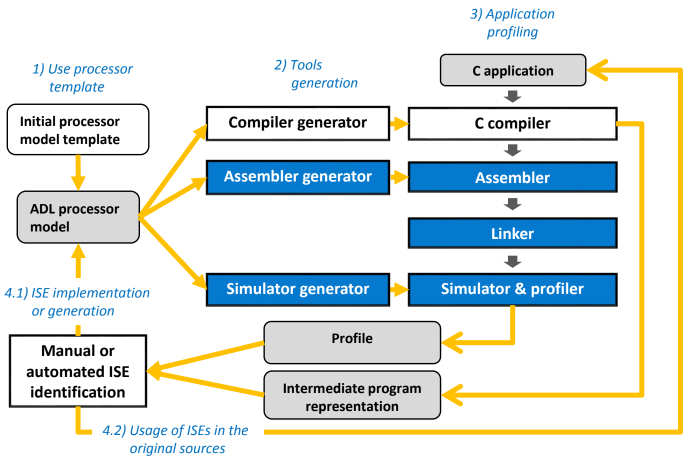

3) Application profiling
1) Use processor template
2) Tools generation
C application
Initial processor model template
Compiler generator
C compiler
Assembler generator
Assembler
ADL processor model
Linker
4.1) ISE implementation or generation
Simulator generator
Simulator & profiler
Profile
Manual or automated ISE identification
Intermediate program representation
4.2) Usage of ISEs in the original sources

Figure 1.1: Manual or automated processor optimization starting from a processor template; boxes with white background show components that were missing at the start of work on the dissertation thesis

Chapter 2

State of The Art

2.1 Application Specific Instruction Set Processor Design Tools

Embedded systems are often constrained by power, performance, and cost. For one embedded device a simple 8-bit microcontroller suffices, but other embedded systems must employ multi-core processors with powerful accelerators.

To create an embedded system that conforms to the given requirements, a correct mix of processor cores and accelerators must be chosen.

A general-purpose processor, ASIP, FPGA or ASIC can be used to implement an embedded system. As shown in the following figure 2.1, these options differ in efficiency with which they execute the target application.

2.2 CodAL Language

The CodAL [1] language is a mixed ADL, which allows describing both architectural information for C compiler generation and microarchitectural information for HDL generation.

The processor core can be in the CodAL ADL described on two levels of abstraction: instruction-accurate and cycle-accurate. The instruction-accurate model is very light-weight, allows fast design space exploration, and

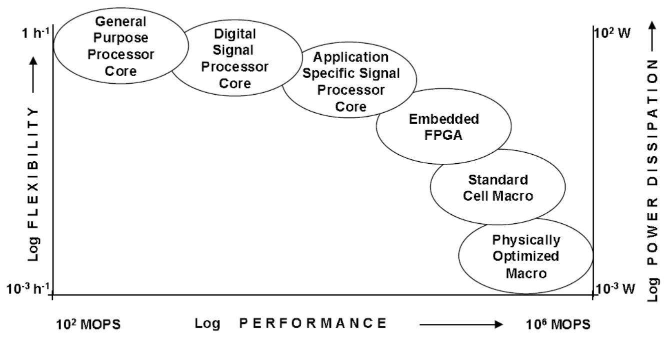


Figure 2.1: Trade-off between flexibility, performance and power consumption [5]

very fast functional simulator generation. New instructions can be added in several minutes without the need to consider the microarchitecture. Also, the behavior of a new instruction can be described in an arbitrary C code, which allows just copying the potentially synthesizable application part and using it as the behavior of a new ISE in order to quickly see the results. The cycle-accurate model describes the processor's pipeline, is used for processor synthesis to VHDL, and may contain specific optimizations for hardware implementation.

Processor resources, instruction set syntax and binary coding description can be shared between these two models.

This approach with two different abstraction models allows fully automatic equivalence checking of instruction-accurate and cycle-accurate models either through bounded model checking approaches [6] or by using functional verification [21]. This way the instruction-accurate model can be seen as a golden model, and the cycle-accurate model is a more concrete refinement of it. Also multiple cycle-accurate models (optimized for speed, area, or power consumption) may exist for one instruction-accurate model.

The CodAL language is supported by Codasip Framework tools [7]. Codasip Framework is an EDA (Electronic Design Automation) tool for fast ASIP (Application-Specific Instruction Set Processor) design. Using the Eclipse-based Codasip Studio graphical interface, the user defines models in CodAL. Generated from the processor model can be the C compiler, assembler, simulators with different accuracies and profiling levels with debugging capabilities, VHDL description, and verification supporting tools.

2.3 High Level Language Compilers

In this section, we will review the problematics of retargetable compilers. The full thesis text also contains detailed analysis of existing retargetable compilers, reconfigurable cores, and instruction set optimization techniques.

2.3.1 Higher Language Level Compiler Structure

Higher Language Level (HLL) compilers generally follow a 3-part structure as shown in figure 2.2. First a *frontend* processes the input code, then usually an architecture independent *optimizer* optimizes the code, and a target-dependent *backend* transform the target-independent intermediate representation (IR) into assembly code.

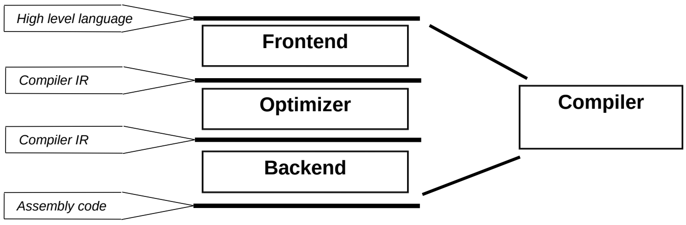


Figure 2.2: 3-part HLL compiler structure

Since we need to generate a compiler, we will focus mainly on the target-dependent part of the compiler, i.e. the *backend*. Retargetable compilers differ from other compilers by clearly separating the target-dependent and target-independent components, and by providing means for modifying the target-dependent components.

2.3.2 Retargetable Compilers

A retargetable compiler can be classified either as [13]:

- *parametrizable*, where the machine description consists of only numerical parameters and subtarget settings,

- *user-retargetable*, where the external machine description given in a dedicated language contains the retargeting information and its specification does not require in-depth compiler knowledge, or
- *developer-retargetable*, where the target architecture description is also mostly in external files, but its specification requires extensive compiler expertise.

It is up to the processor design tool developers to make the compiler retargeting based on an ADL model as user-retargetable as possible. This involves automating of all the tasks that can be reasonably automatized, and also providing more abstract and user-friendly definition languages for the definition of the remaining compiler features while hiding compiler implementation details.

2.4 State of the Art - Conclusion

The planned goal of this thesis, as was described in the introduction, is to create C compiler generator, an extensible ASIP processor template, and methods how to optimize the processor template using the compiler by adding new instructions. This, together with other tools, will form a complete ASIP design and optimization environment, where a user starts with a processor template and optimizes it for his/her needs.

For C compiler generation, it is necessary to use an existing compiler as a base platform, because to develop a compiler is a complex, and long-term task for large teams. The best 2 retargetable compiler platforms with the highest potential to be still improved in the future are LLVM, and GCC. We chose LLVM, because of its newer, and cleaner implementation, and a more permissive license.

The only commercially used compiler generators based on ADLs use languages nMl, and LISA. Compiler generation from nMl is not published. In the LISA compiler generator, they described the instructions for the compiler generator with a special language different from the language used to define instructions for the simulator. This can introduce inconsistencies in the instruction accurate LISA model, and also makes the ADL language, and model creation more difficult. The goal is to find an approach that can use just one description. This would the mean that the ADL is kept simpler, and the inconsistency problem is avoided.

Existing reconfigurable processor cores such as Tensilica Xtensa, or Synopsys ARC that use a fixed instruction set that can be extended with configurable options such as floating point, or DPS instructions. This limits the explorable design space, because the user of these extensible processor is limited in the changes he can make. E.g. it may be necessary to add reading ports to the register file, or to modify the memory interfaces. This is not possible neither with Xtensa, nor with ARC. We already have an ADL language that allows arbitrary modifications to the processor, so we are not limited this way. The goal for the new extensible core is a very efficient base instruction set. Parts of an application can be accelerated with ISEs, but the rest will run using standard instructions. It is also important that the new processor will be easily extensible. The microarchitecture must be rather simple, so the user can understand it and make modifications to it.

Finally, the ASIP optimization approaches can be roughly divided into automatic, and manual. Pure automatic methods allows only limited speedup. On the other hand, to employ manual methods is very time consuming. The goal will be to find a method that allows the user to choose his own extensions, and automatize his/her efforts as much as possible.

We refined the goals of this thesis with respect to the current state of the problematics. The next chapter describes how were these goals solved.

Chapter 3

Solution

This chapter contains description of the solution done by the author with the goal of providing tools for reconfigurable processor design.

The main contribution presented here is a process of transformation of a CodAL model into a form usable by the C compiler generator. Tools that do this transformation were designed and majority of them was implemented by the author. This process of transformation is done a tool *Semantics extractor* and is described in section 3.1.

This is followed by C compiler backend generator description that was designed by the author. The *Backend generator* is covered in section 3.2. Overall structure of the whole compiler generator is shown in figure 3.1.

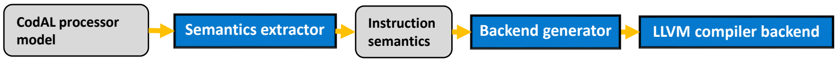


Figure 3.1: Overall structure of the C compiler generator

To be able to test the generated C compiler and also to perform experiments with reconfigurable processors, the author designed five extensible processor cores and implemented three of them as instruction accurate CodAL models. The processor cores are described in section 3.3.

Finally, a method is shown that helps the user identify new instructions, and to move them automatically into a template of an extensible processor core.

3.1 Instruction Semantics Extraction

For compiler backend generation, the compiler generator needs as input some kind of analyzable model of instruction set description. The developed model is called *instruction semantics* model.

First, we specify some requirements on the instruction semantics model with respect to compiler generation. The compiler generator must be able to identify several important instructions, the most significant are: register moves; memory accesses, also with accesses to a stack; load of a pointer-wide (e.g. 32-bit) immediate value for global addresses; elementary operations needed during instruction selection and operation lowering; and no-operation instruction for hazard-free scheduling, jump delay slots and other scheduling purposes.

The compiler generator also needs the behavior of instructions in a DAG-like (directed acyclic graph) form to generate instruction selection patterns such as the one in figure 3.2. The instruction semantics model must also contain information about processor registers and register operands to supply information for register allocator. Furthermore, the instruction set model must be simple and precise enough to do these analyses quickly and unambiguously.

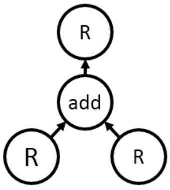


Figure 3.2: Desired form of instruction semantics representation as a DAG of and addition instruction

The main components of the Semantics extractor are shown in figure 3.3. First are all instruction enumerated by the tool *semextr*. Then in more than 40 steps is the output of the *semextr* processed and optimized. Finally the output contains a list of instructions in their simplest possible form usable for compiler generation.

As an example, we will show an addition instruction. The desired form required by the compiler generator is shown in figure 3.2.

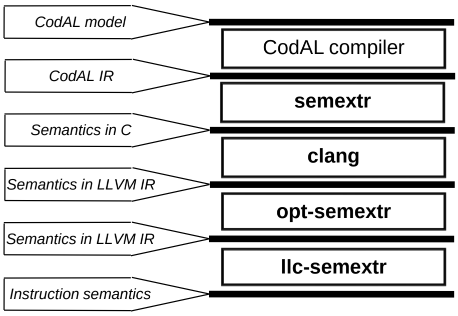


Figure 3.3: The main components and intermediate representations in the Semantics extractor

And you can see the output of the semantics extractor here:

```
instr i_3_reg_operands__opc_add__gpreg__gpreg__gpreg__,
    ok,
  // Instruction operands
  { gpreg_0 = regop(gpreg), gpreg_1 = regop(gpreg),
    gpreg_2 = regop(gpreg) },

  // Semantics
  %u0 = i32 gpreg_1;
  %u1 = i32 gpreg_2;
  %add = add(%u1, %u0);
  gpreg_0 = %add;

  ,
  // Syntax
  "ADD" gpreg_0~"," gpreg_1~"," gpreg_2,
  // Binary coding
  0b000100 gpreg_0[4,0] gpreg_1[4,0] gpreg_2[4,0]
    0b00000000000
```

The `gpreg_1` and `gpreg_2` are input register operands, their values are added, and the is the result stored into an output register operand `gpreg_0`. This is just another representation of the DAG from figure 3.2 needed by the compiler generator.

3.1.1 Semantics Extraction Results

Semantics extractor works for many diverse architectures and when the CodAL model follows certain simple guidelines such as explicitly marked register classes, and flag registers modeled as 1-bit registers, it outputs the instruction semantics in a very simple form.

Semantics extractor supports diverse features present in current architectures, such as floating point with diverse special floating point operations, SIMD instructions, indexed register access, and saturated operations. It is also possible to define subinstructions, and user functions for often repeated parts of code.

Results for diverse architectures are presented in table 3.1. These results were obtained for Semantics Extractor version 2.1.7 based on LLVM 3.4. The architectures were modeled as Instruction Accurate CodAL models, the models that were implemented by the author are marked with an asterisk. The next column contains the count of instructions generated by the first step of semantics extraction, the *semextr* tool that generates all possible instructions from a CodAL model. Size in bytes contains resulting size of the `compiler_semantics.sem` file. To show relative size needed per one instruction, the Bytes per instruction column contains file size divided by the number of instructions. The size per instruction varies between 336 and 553 bytes (with the exception of the Intel 386 architecture that has in average more complex instructions). Architectures whose arithmetic instructions set flags such as carry, overflow, etc. have higher average size per instruction, because setting of these flags is included in the instruction semantics.

Finally, the last column contains runtime of the Semantics extractor built with gcc version 4.8.3 with optimization O3 on a PC with an operating system Linux Fedora 20, processor Intel Core i7-4770 CPU running at 3.40GHz, 16GB of RAM, and an SSD hard disk.

Architecture	Instrs.	Size in ytes	Bytes/instr.	Time
ADOP	537	262050	488	1.96 s
ARM7 cc*	532	316374	595	8.27 s
AVR32	1521	624932	411	5.53 s
Codix Experimental*	3025	1133815	375	10.48 s
Codix Risc VLIW*	2588	995735	385	9.36 s
Codix Risc*	2585	977172	378	8.01 s
Codix Stream*	808	385088	477	4.05 s
Codix uRisc*	40	17504	438	0.38 s
Infineon Tricore	212	110344	520	1.66 s
Intel 386	10079	9481782	941	283.18 s
Microblaze	481	170849	355	1.69 s
MIPS basic*	308	105396	342	1.35 s
MIPS*	462	178678	387	3.30 s
Open RISC	241	81081	336	1.07 s
Power PC	177	97838	553	1.41 s
Simple Flag	47	23092	491	0.49 s
Simple Flag Float	74	34601	468	0.59 s
Vix	3348	1412534	422	32.05 s

Table 3.1: Semantics Extractor instruction counts, resulting file size, and runtimes for diverse CodAL models

The instruction semantics format was also successfully used for fast simulator generation [17], in a reverese compilation tool [12], and in verification [6]. Semantics extraction was published by the author in [10], and in [11].

The main advantage of the Semantics extractor is its ability to convert an instruction accurate CodAL model into a model suitable for compiler generation. This way, instruction behavior can be described only once, and no inconsistencies can occur e.g. as in the LISA ADL approach, where they have 2 descriptions - one for simulator, and one for compiler.

Also using a compiler to generate itself is very useful, because the resulting form of instructions and ordering of operations is exactly the same that then appears during compilation. This assures that instruction selection patterns generated from the instruction semantics match the compiled code.

With strong optimizations and additional transformations are instructions, originally described with complex C code, optimized into their into their simplest possible, and also canonical form. This greatly simplifies analyses over the instruction semantics model.

3.2 Retargetable LLVM Compiler Generation

The Semantics extractor that we just described converts the CodAL model into a representation suitable for C compiler generation. From this description is generated a C compiler backend. The design of the *Backend generator* is described in this section.

The author's contribution to the compiler generator was its overall design, preparation of many models for compiler generator testing, collecting and preparation of tests, lead of the implementation works, and setting priorities on new features, optimizations, and stability. Majority of implementation works and internal components design was done by Jan Hranáč. Testing and its automation was done by Luděk Dolíhal, he also was the main quality engineer for the compiler generator. Additional optimizations were done as a part of numerous bachelor and master theses led by the author.

3.2.1 Overal Compiler Generator Design

Structure of the Backend generator is shown in figure 3.4 and has following inputs from the user:

- *Instruction semantics* file is generated by the Semantics extractor, this is the only required input, information contained in other inputs can be automatically inferred from the instruction semantics file. User may provide additional inputs when he needs to override the automatically inferred values.

- *User semantics* file may contain:

  - ABI (Application Binary Interface) definition such as register usage and calling convention,
  - explicit setting of flag registers such as carry, overflow, etc. (if they cannot be detected automatically),

  - settings for the instruction scheduler, e.g. whether the compiler should handle structural or data hazards, and
  - user instruction aliases, for example to describe an existing instruction with equivalent, but slightly different semantics so the Backend generator can recognize such instruction when the automatically generated semantics is not suitable.

- *User instruction equality rules*: equality rules are mainly used to specify combinations of instructions used to perform comparisons, conditional selects, and conditional jumps. Some architectures use unusual flags, or miss several comparison types, so with these equality rules the user can define what instructions should be used e.g. for floating-point equality comparison. The Backend generator already contains a huge set of these equivalencies and can use them automatically (the *Default instruction equality rules*).

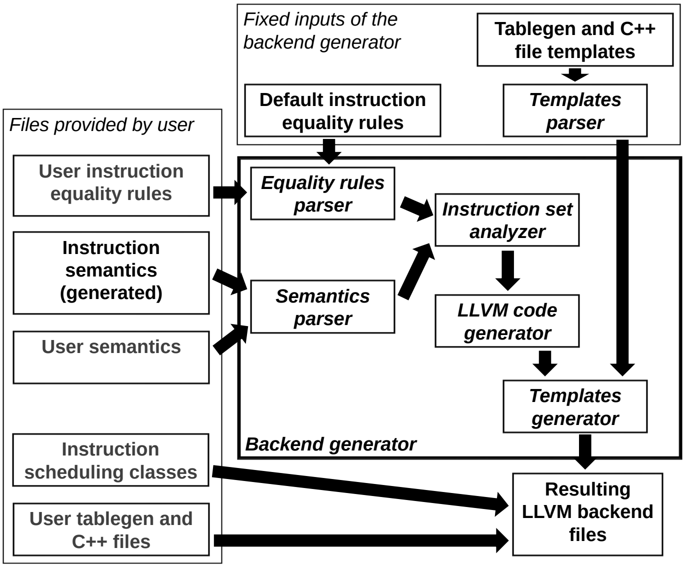
- *Instruction scheduling classes* may override automatically generated instruction schedule. Standard LLVM definition for the LLVM *tablegen* tool is used and this allows for example to specify that a load from memory has latency 3 cycles and then uses a shared register file. The LLVM scheduler can then reorder instructions to minimize structural and data hazards.

- *User tablegen and C++ files* are used to override and extend automatically generated LLVM sources. For example, the Backend generator may not be able to detect how to store some special (e.g. flag) register to memory. By using virtual methods the user can simply extend the generated sources to perform such operation.

Another input for the Backend generator are Tablegen and C++ file templates. This input can be modified by a Backend generator developer and should not be usually modified by the user.

3.2.2 Generated Compiler Results

The Backend generator currently generates architecture files for LLVM version 3.4. The Backend generator can fully automatically generate a backend for architectures listed in table 3.2. It is also automatically tested on these architectures on a set of tests in C language from GCC torture testsuite, LLVM Testsuite, and a set of in-house developed tests.



Fixed inputs of the backend generator
Tablegen and C++ file templates
Default instruction equality rules
Templates parser
Files provided by user
User instruction equality rules
Equality rules parser
Instruction set analyzer
Instruction semantics (generated)
Semantics parser
LLVM code generator
User semantics
Templates generator
Backend generator
Instruction scheduling classes
Resulting LLVM backend files
User tablegen and C++ files

Figure 3.4: Inputs and main components of the backend generator

For architectures where a standard C library Newlib is available is the number of tests that are run higher. Many tests rely on the `int` data type to have 32 bits, so the 16-bit architectures whose `int` data type has 16 bits have more fails. Some tests check some obscure cornerstones or undefined behaviors of the C language, so there were always some failing tests. If the failing tests value is lower than 1% (or around 10% for 16-bit architectures), the compiler can be regarded as stable and can compile any complex application correctly. The table contains values from testing obtained on 20th of August 2014.

Architecture	**Tests**	**Failing tests (%)**	**Comments**
ADOP	1688	11.0	16-bit
ARM cc	1688	3.7	
AVR32	1688	3.9	
Codix Experimental	2392	1.3	Newlib
Codix RISC	2392	0.8	
Codix STREAM	1677	10.0	16-bit
Codix URISC	2392	0.7	Newlib
MIPS	2392	1.2	Newlib
MIPS basic	2392	0.7	Newlib
Open RISC	1688	0.9	
Simple Flag	2392	0.8	Newlib
Simple Flag Float	2392	1.0	Newlib
Vix	2392	0.8	Newlib

Table 3.2: Results of generated compiler testing

LLVM compiler backend files are generated by the Backend generator in several seconds, and then building the backend takes approximately one minute on a standard PC.

We will now compare the generated code performance on the MIPS version Release 1 architecture. For this comparison were following compilers used: the generated LLVM 3.4-based compiler from a MIPS CodAL model (further only GEN LLVM), then hand-written compiler for MIPS present in LLVM 3.4 (MIPS LLVM), and MIPS GCC 4.9 (MIPS GCC). Source codes of applications come from the LLVM testsuite, that is a collection of tests and many diverse benchmarks.

Relative performance is shown in graph 3.5. As a baseline was taken count of simulated clock cycles for the GEN LLVM. Percentage values for MIPS LLVM, and MIPS GCC were obtained by dividing the number of clock cycles for the generated compiler by clock cycles for MIPS LLVM, resp. for MIPS GCC. Higher percentage means higher relative performance, i.e. higher is better.

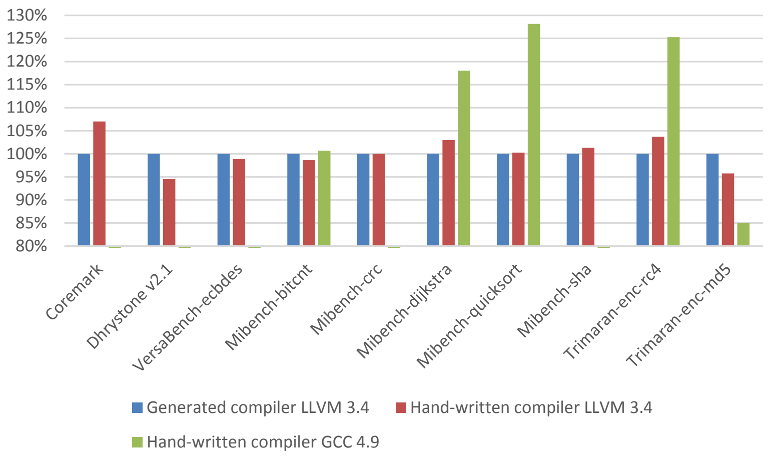


Figure 3.5: Comparison of compiler performance for the MIPS architecture

You can see that the performance of the GEN LLVM compared with MIPS LLVM is very close, and in some cases even better than the hand-written compilers. Differences between MIPS LLVM and MIPS GCC are caused by different optimizations mainly in the architecture-independent optimizers of these compilers.

Corresponding relative code size for these benchmarks is shown in figure 3.6. Code sizes for Mibench-bitcnt and Mibench-crc are very small (approx. 200 bytes), so the relative difference is high. Lower percentage means lower code size, i.e. higher is worse.

Results for the Codix RISC processor core are also presented in the following section 3.3.

Semantics extractor together with Backend generator perform many tasks fully automatically, and can generate C compiler backend for many architec-

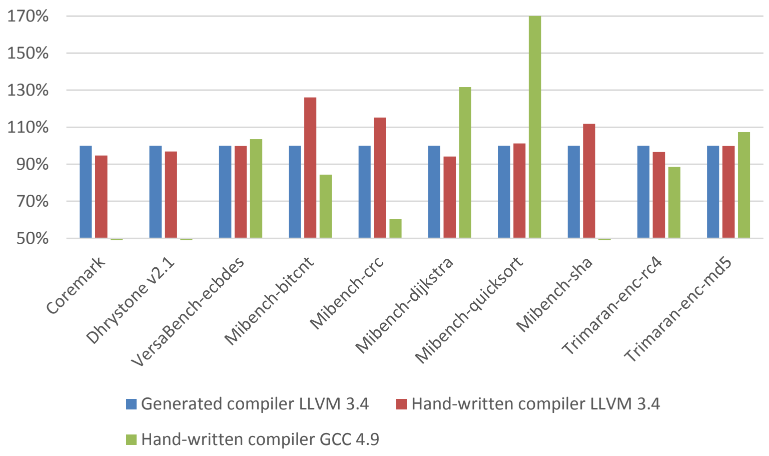


Figure 3.6: Comparison of code size for the MIPS architecture

tures. To create a compiler backend by modifying the backend sources can take more one month even to someone that knows the retargetable compiler infrastructure. On the other hand, to create a CodAL model, and to generate a backend with the developed Semantics extractor, and Backend generator can be done in several days. This is an enormous boost in designer productivity. The solution that was developed is also fully automatic, so the user does not need to know about the LLVM internals. Only when some specific optimizations are needed, the user can add some extension manually.

Compared to the only published and commercially used solution that uses the LISA ADL language, the biggest advantage is that the CodAL model needs just one description of the instruction semantics, and there is no need to modify the LLVM sources.

Another big advantage that the compiler is generated automatically is the ease of adding, and removing instructions. New instruction, if an instruction selection pattern can be generated for it, can be used fully automatically by the C compiler. Also a very fast design space exploration, and optimization of the instruction set can be made. For example the user can decide to optimize on area by removing instructions, he/she then only regenerates the compiler, and runs benchmarks.

Despite the compiler's automatic generation, it provides performance comparable with hand-written solutions. Also several extensions, especially for VLIW architectures, and SIMD instructions were implemented. These extensions then can provide even higher performance than existing compilers.

3.3 Reconfigurable Processor Core Codix RISC

This chapter describes one of the processor cores designed by the author called Codix RISC and modeled in the Codasip Framework as instruction accurate CodAL model. The author also designed the high-level microarchitecture of the processor core, the detailed implementation of cycle accurate CodAL models for Codix RISC was done by Marián Pristach from The Faculty of Electrical Engineering and Communication, Brno University of Technology.

Instruction Set

Codix RISC has 32 32-bit general purpose registers, and 1 special status register. The general purpose register `r0` is always zero. Memory used 32-bit words with little endian ordering.

The operations in the instruction set, and also their syntax is based on LLVM IR. This then helps better matching of IR operations to particular instructions.

One feature that was inspired by ARM is shifting an input register operand. Second source register operand can be shifted in Codix RISC by 0, 1, 2, or 3 bits to the left, as e.g. in `r1 = add r2, shl2 r3`. In ARM, the shift amount can be a value from 0-31, but from profiling results, most of the shift amounts are never used and this then forms a critical path between through the ALU, shifter, and forwarding logic. The solution used in Codix RISC does not need a full shifter, only a 4-input multiplexer suffices. Additional operations over the second register operand can be easily added.

To compute an address for a load or store can be 2 addressing modes used: register + signed offset, or register + shifted register.

Another difference to MIPS, or ARM, is that conditional jumps can perform comparison of 2 registers, including binary and logical operations. In ARM, flags are used for conditional jumps, and MIPS provides only comparison on equality, unequality, and comparisons with zero.

Interrupts are also supported in Codix RISC. One bit in the status register specifies whether interrupts are enabled, and instructions to call an interrupt, and to return from interrupt are provided.

Area and Performance

Codix RISC synthesis results are shown in table 3.3. Synthesis was done with Xilinx ISE WebPack 14.2 with VHDL code generated from Codix cycle-accurate model with Codasip Framework 1.8.1 [7]. Family is particular Xilinx FPGA type, Speed specifies speed grade selected for synthesis, LUTs are used lookup-tables, Flip Flops specify used 1-bit memories and fmax is maximal frequency recommended for the design by synthesizer.

Family	**Speed**	**LUTs**	**Flip Flops**	f_{max} **[MHz]**
Spartan3	-5	3145	570	56.213
Spartan3E	-5	3102	566	65.016
Spartan6	-3	1835	594	72.163
Virtex5	-3	1805	564	140.443
Virtex6	-3	1853	572	159.571
Kintex7	-3	1871	567	172.655

Table 3.3: Synthesis results for extensible processor core Codix RISC

Codix RISC was also synthesized for the 40-nm TSMC technology. Without caches, it can run on 450 MHz. When synthesized to 500 MHz with Synopsys Design Compiler, the total cell area is 40441 μm^2 (0,04 mm^2) (also without caches).

In table 3.4 is Codix compared with other soft-processors using the Coremark benchmark. Coremark is an application measuring pipeline throughput [8]. The Coremark benchmark was compiled with automatically generated C compiler based on LLVM 3.0 with optimization -O3 and simulated on an cycle accurate simulator (with simulated 8kB 4-way caches for instructions and for data) using Codasip Framework v. 1.8.1. The Coremark/MHz value is calculated as 1 000 000 divided by the number of cycles needed for one iteration of the Coremark benchmark. Results for the other processor cores in table 3.4 come from measurements described in article [4].

The Codix RISC architecture was also compared to the Microblaze, ARM,

Processor Core	Compiler	Coremark/MHz
Codix RISC	LLVM 3.0	1.65
Leon 3	gcc 4.4.2	1.91
MicroBlaze	gcc 4.1.2	1.90
OpenRISC	gcc 4.5.1	1.38
Nios II	gcc 4.1.2	1.93
TI Omap 3430 (ARM Cortex A8)	gcc 4.7.0	2.24

Table 3.4: Comparison of soft-processors for the Coremark benchmark running on hardware

and ARC architectures using Dhrystone 1, Dhrystone 2.1, and Coremark benchmarks. The clock cycles needed to execute each of the benchmarks were counted using instruction accurate simulators. To simulate Codix RISC was used Codasip intersim 3.0.1, for other cores were used simulators from Open Virtual Platforms build 20130630 [2]. To make the comparison fair, data hazard handling for Codix RISC was disabled in the compiler, because the other simulators are instruction accurate, and do not count stalled cycles. Disabling data hazard handling in compiler disables generation of additional NOPs, mainly for load instructions. All benchmarks were compiled with -O3 optimization level. Following compilers were used: arm-gcc 4.8.1, microblaze-gcc 4.8.1, arc-gcc 4.8.0, codix-risc-llvm 3.2 (codasip), and arm-llvm 3.2. The benchmarks did not need any standard C library to be linked and executed. Comparison is shown in graph 3.7, ther comparisons are in the full text of the dissertation. The WPO values are results when using whole program optimization in LLVM. With WPO is the whole application linked together at the LLVM IR level, and additional intraprocedural optimizations (IPO) such as inlining. Results for Microblaze with Coremark was omitted, because it did not match the results from table 3.4 (it was on 40% of performance of other cores), assuming that there was a mistake in measurement.

The last comparison presented here is for the FFMEG application, which is a complex application having 30MB of source codes. Codix RISC was compared where compared to ARMv7 with compiler GCC 4.8. It takes 6% more cycles on an instruction accurate simulator with Codix RISC to decode MPEG4 video compared to ARM.

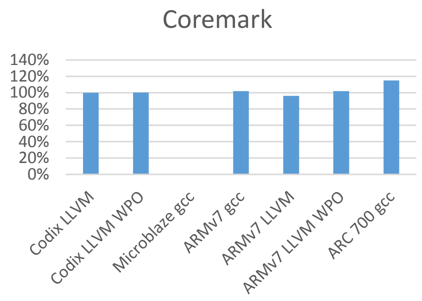


Figure 3.7: Comparison for Coremark on simulators

Codix RISC - Conclusion

The Codix RISC processor was successfully used to run many applications including operating system Linux, and environment .NET Micro framework. Also applications using the OpenCV library can be run on it. Without any extensions, it provides performance comparable to other widely used processor cores. Its biggest advantage is its extensibility, where compared for example to Tensilica Xtensa, or Synopsys ARC, any part of the processor pipeline can be changed, so the user has much more freedom to do diverse optimizations.

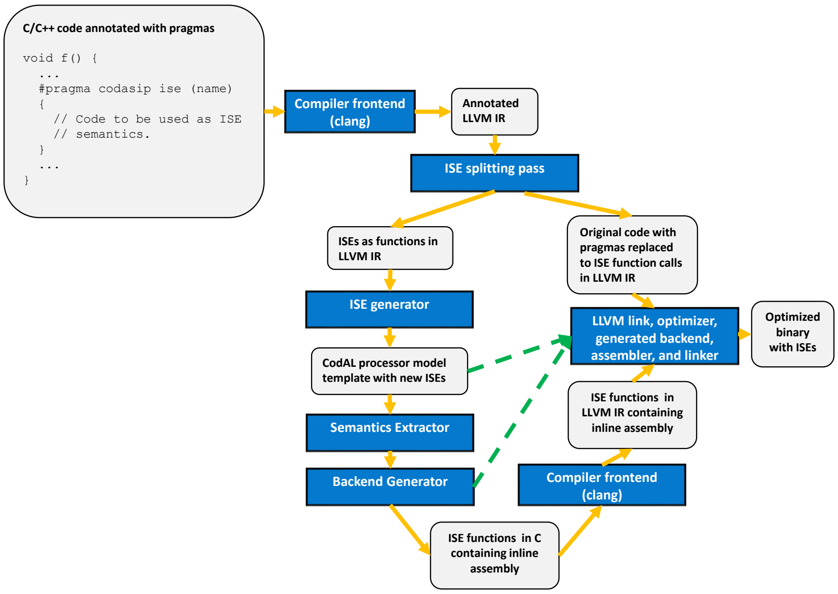
At the time of finishing this thesis was this processor evaluated by Exar Corporation. At Exar, they plan to replace an ARM processor in their design for surveillance cameras. A competition to Codix RISC is the Synopsys ARC processor. Their decision is not known yet.

3.4 Manual ISE identification

Two bachelor theses on automatic Instruction Set Extension (ISE) identification were led by the author. First one used algorithm *single cut* [15], the second used algorithm *ISEGEN* [20]. These theses were quite successful, but the obtained speedup was rather limited.

Also, with regards to other published results on automatic ISE identification [22], the author decided to focus mainly on automation of user-guided identification. The scheme of this user support is shown in figure 3.8.

The user first marks interesting parts of code with pragma `codasip ise`. The C code is the compiled by a frontend to LLVM IR, pragmas are kept


C/C++ code annotated with pragmas
Compiler frontend (clang)
Annotated LLVM IR
ISE splitting pass
ISEs as functions in LLVM IR
Original code with pragmas replaced to ISE function calls in LLVM IR
ISE generator
LLVM link, optimizer, generated backend, assembler, and linker
Optimized binary with ISEs
CodAL processor model template with new ISEs
ISE functions in LLVM IR containing inline assembly
Semantics Extractor
Backend Generator
Compiler frontend (clang)
ISE functions in C containing inline assembly

```
void f() {
  ...
  #pragma codasip ise (name)
  {
    // Code to be used as ISE
    // semantics.
  }
  ...
}
```

Figure 3.8: Scheme of support for manually identified ISEs

in the LLVM IR as code annotation. Then an ISE splitting pass is run. It removes annotated block of code from the application and replaces them with calls to functions with name prefix `__ise`, the result is stored in file `prog.ll`. Another LLVM IR file `ises.ll` is created that contains the removed code blocks as functions. This file `ises.ll` is processed by the ISE generator that generates one instruction for each of the ISE function in the CodAL model template.

This model is then used to generate compilation tools, and the Backend generator also generates a file `inlines.c` containing functions with inline assembly. The replaced code blocks are usually too complex to be matched by the instruction selector, so using inline assembler is necessary.

The file `inlines.c` is then compiled with a frontend into a LLVM IR file `inlines.ll`. Then it is linked together with `prog.ll`, and optimized. The optimizations inlines the functions from `inlines.ll` into call sites from `prog.ll`, so that no overhead of function calls is present in the resulting program. Optimized file is then compiled, assembled, and linked, and an optimized binary is created. The CodAL model template also contains definitions of the new ISEs.

The resulting binary is then simulated. If the performance with new ISEs is not sufficient, the user can change the pragmas in the original source code, and try a new optimization opportunity.

Support for pragmas in *clang*, and the tool ISE generator was implemented by the author. As a processor template was used the Codix RISC model. The ISE splitting pass was implemented in a bachelor thesis [16] led by the author. The student also made this whole infrastructure work, and added other necessary transformations and externsions such as lookup tables support. There are currently not many results on using this infrastructure, because it was finished only recently, from some quick tests, speedups from 1.5 to 7.8 for different benchmarks can be obtained very quickly.

Chapter 4

Conclusion

In the introduction was as a goal planned to create a complete environment for optimization of processor cores with using processor templates.

The largest part of this goal was to create a compiler generator. With Semantics Extractor, and Backend generator, a LLVM-based compiler can be automatically generated from a CodAL model. The performance of the code produced by the generated compiler is comparable with hand-written solutions, and in some cases even produces faster code. Also many extensions, mainly for VLIW architectures were implemented that are better than existing solutions. The resulting compiler is used commercially in the Codasip spin-off that originated from the Lissom project.

The author designed and implemented the Semantics Extractor with its output format, and led the works on the remaining parts. One of the main original contributions is the process used in the Semantics Extractor, where a compiler is used to generate itself. Also, compared to other existing solutions (e.g. LISA ADL), just one instruction semantics representation is needed. As a result is produces an easily analyzable model of the instruction set that has shown to be useful also for other areas such as fast simulator generation, reverse compilation, and verification.

The author also designed several processor architectures with Codix RISC currently being the most useful one. In its base configuration without extensions, the performance is comparable to other existing processor architectures. The Codix RISC processor can run Linux, OpenCV, and other complex applications that can be compiled with the generated compiler. This shows both the quality of the compiler, and the processor core. Codix RISC is currently commercially offered by Codasip as an extensible processor core. Compared to the Synopsys ARC processor tools, the compiler for Codix RISC is generated automatically, and can take advantage of newly added instructions.

Results were published in numerous papers, and the author also co-authored 2 US patents [19] owned by Brno University of Technology, and [18] owned by Codasip. Both patents are currently pending.

Finally, the scheme shown in figure 3.8 (in the previous section) is in fact the tool for fast design space exploration, and optimization of ASIPs that was envisioned as the goal of this dissertation. Using a compiler generator, processor template, and other tools, the author with the help of other people created a tool for very efficient ASIP optimization.

Bibliography

[1] CodAL manual. Codasip, Brno, CZ, 2014.

[2] Open Virtual Platforms (OVP) portal, [online] http://www.ovpworld.org/ (August 2014).

[3] Project Lissom Webpages, [online] http://www.fit.vutbr.cz/research/groups/lissom/ (August 2014).

[4] Sven-Ake Andersson: Four soft-core processors for embedded systems, EETimes, 2013.

[5] Blume, H., Hübert, H., Feldkämper, T., Noll, T. G.: Model-Based Exploration of the Design Space for Heterogeneous Systems on Chip. In *ASAP '02: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, page 29, 2002.

[6] Charvát, L., Smrčka, A., Vojnar, T.: Automatic Formal Correspondence Checking of ISA and RTL Microprocessor Description. In *Proceedings of the 13th International Workshop on Microprocessor Test and Verification (MTV 2012)*, pages 6–12. Institute of Electrical and Electronics Engineers, 2012.

[7] Codasip. Codasip Framework Tools, [online] https://www.codasip.com/products/tools/ (August 2014).

[8] EEMBC. CoreMark an EEMBC Benchmark, 2014.

[9] Adam Husár: Implementace obecného assembleru. Master's thesis, FIT, Brno University of Technology, Brno, 2007.

[10] Husár, A., Hruška, T., Trmač, M., Přikryl, Z.: Instruction selection patterns extraction from architecture specification language isac. In *Proceedings of the 16th Conference Student EEICT 2010 Volume 5*, pages 166–170. Faculty of Information Technology BUT, 2010.

[11] Husár, A., Trmač, M., Hranáč, J., Hruška, T., Masařík, K., Kolář, D., Přikryl, Z.: Automatic c compiler generation from architecture description language isac. In *6th Doctoral Workshop on Mathematical*

and Engineering Methods in Computer Science, pages 84–91. Masaryk University, 2010.

[12] Křoustek J., Pokorný, F.: Reconstruction of instruction idioms in a retargetable decompiler. In *4th Workshop on Advances in Programming Languages (WAPL'13)*, pages 1507–1514. IEEE Computer Society, 2013.

[13] Leupers, R., Marwedel, P.: *Retargetable compiler technology for embedded systems: tools and applications*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[14] Maxfield, C.: Wow! Tensilica licensees have shipped 2 billion IP cores!, 2012.

[15] Melo, S.: Zrychlení vykonávání softwaru pomocí automatických instrukčních rozšíření. Bachelor's thesis, FIT, Brno University of Technology, Brno, 2013.

[16] Mikó, A.: Semiautomatická optimalizace pomocí specializovaných instrukcí. Bachelor's thesis, FIT, Brno University of Technology, Brno, 2014.

[17] Ministr, M.: Virtuální platformy pro simulaci instrukčních sad. Bachelor's thesis, FIT, Brno University of Technology, Brno, 2014.

[18] Přikryl, Z., Husár, A., Masařík, K., Hruška, T.: A method and an apparatus for automatic processor design and verification, US patent, pending, owner: Codasip ltd., 2014.

[19] Hruška, T., Přikryl, Z., Husár, A.: A method and an apparatus for instruction set translation using finite state automata, US patent, pending, owner: Brno University of Technology, 2013.

[20] Česka, M.: Automatické vyhledávání instrukčních rozšíření aplikačních procesorů. Bachelor's thesis, FIT, Brno University of Technology, Brno, 2013.

[21] Šimková, M., Přikryl, Z., Hruška, T., Kotásek, Z.: Automated functional verification of application specific instruction-set processors. *IFIP Advances in Information and Communication Technology*, 4(403):128–138, 2013.

[22] Galuzzi. C., Bertels, K.: The instruction-set extension problem: A survey. *ACM Trans. Reconfigurable Technol. Syst.*, 4(2):18:1–18:28, May 2011.

Curriculum Vitae

Personal Data

Name:	Adam Husár
Born:	January 22, 1983
E-mail:	ihusar@fit.vutbr.cz
Homepage:	http://www.fit.vutbr.cz/∼ihusar/

Education

- 2007 Ing., Faculty of Information Technology, Brno University of Technology
- 2005/2006 Erasmus internship, ESIEE Amiens, France
- 1999 Gymnázium Mariánské Lázně
- Foreign languages: fluent in English and French, passive knowledge of German, basics of Spanish and Russian

Experiences

- 2012 – 2014: Compiler group leader, Codasip ltd., and ApS Brno ltd. Codasip Division, CZ
- 2010 – 2012: Compiler developer, ApS Brno ltd. Codasip Division, CZ
- 2009: Instruction set simulator developer, OnDemand Microelectronics A.G., Vienna, AU
- 2007: Embedded systems programmer, Honeywell, Brno, CZ

- 2006: Mobile applications programmer, Erasmusse, Amiens, FR

Research Projects

- MPO FT–TA3/128: *Language and Development Environment for Microprocessor Design* (2005 – 2009) - research in the field of processor programming tools.
- MPO FR–TI1/038: *System for Programming and Realization of Embedded Systems* (2009 – 2011) - research in the field of processor programming tools.
- SMECY - Artemis JU: *Smart Multicore Embedded SYstems* (2010 – 2012) - research in the field of multicore processors programming.
- PaPP - Artemis JU: *Developing Future-Proof Parallel Software* (2010 – 2012) - research in the field of multicore processors programming.

Patents

- Přikryl, Z., Husár, A., Masařík, K., Hruška, T.: A method and an apparatus for automatic processor design and verification, US patent, pending, owner: Codasip ltd., 2014.
- Hruška, T., Přikryl, Z., Husár, A.: A method and an apparatus for instruction set translation using finite state automata, US patent, pending, owner: Brno University of Technology, 2013.

Products

- Hruška, T., Husár, A., Masařík, K.: C Language Compiler Frontend with Pragma Support, software, 2014.
- Husár, A., Maršík L.: HW accelerator for radar signal processing, specimen, 2014.
- Hruška, T., Husár, A., Masařík, K.: Robust Automatic Vector Accelerator Compiler, software, 2014.
- Hruška, T., Masařík, K., Přikryl, Z., Husár, A., Fujcik L., Pristach M.: Microprocessor ADOP, specimen, 2011.

Selected Author's Publications

[i] Husár, A., Přikryl, Z., Dolíhal, L., Masařík, K. a Hruška, T.: *ASIP Design with Automatic C/C++ Compiler Generation*. Haifa, 2013.

[ii] Dolíhal, L., Hruška, T., Husár, A., Masařík, K. a Přikryl, Z.: *Use of Architecture Description Language ISAC for ASIP Design*. ACACES 2012, Poster Abstracts. Fiuggi: High Performance and Embedded Architecture and Compilation, 2012. ISBN 978-90-382-1987-5.

[iii] Pristach, M., Husár, A., Fujcik, L., Hruška, T. a Masařík, K.: *Digital Signal Soft-Processor for Video Processing*. In: Electronic Devices and Systems IMAPS CS International Conference 2011 Proceedings. Brno: Vysoké učení technické v Brně, 2011, s. 180-185. ISBN 978-80-214-4303-7.

[iv] Pristach, M., Husár, A., Fujcik, L., Masařík, K. a Hruška, T.: *Digital Signal Soft- Processor for Audio and Video Processing*. ElectroScope. Plzeň: Západočeská univerzita v Plzni, 2011, roč. 2011, č. 4, s. 1-5. ISSN 1802-4564.

[v] Přikryl, Z., Křoustek, J., Hruška, T., Kolář, D., Masařík, K. a Husár, A.: *Design and Simulation of High Performance Parallel Architectures Using the ISAC Language*. GSTF International Journal on Computing. Singapur: Global Science & Technology Forum, 2011, roč. 1, č. 2, s. 97-106. ISSN 2010-2283.

[vi] Husár, A., Hruška, T., Masařík, K. a Přikryl, Z.: *Instruction Pipeline Modeling using Petri Nets*. In: Proceedings of the International Workshop on Petri Nets and Software Engineering - PNSE'10. Universität Hamburg: Technische Universitat Hamburg-Harburg, 2010, s. 163-164. ISBN 978-972-8692-55-1.

[vii] Husár, A., Hruška, T., Trmač, M. a Přikryl, Z.: *Instruction Selection Patterns Extraction from Architecture Specification Language ISAC*. In: Proceedings of the 16th Conference Student EEICT 2010 Volume 5. Brno: Fakulta informačních technologií VUT v Brně, 2010, s. 166-170. ISBN 978-80-214-4080-7.

[viii] Husár, A., Trmač, M., Hranáč J., Hruška, T., Masařík, K., Kolář, D. a Přikryl, Z.: *Automatic C Compiler Generation from Architecture Description Language ISAC*. In: 6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science. Brno: Masarykova universita, 2010, s. 84-91. ISBN 978-80-87342-10-7.

[ix] Přikryl, Z., Hruška, T., Masařík, K. a Husár, A.: *Fast Cycle-Accurate Compiled Simulation*. In: 10th IFAC Workshop on Programmable Devices and Embedded Systems, PDeS 2010. Pszczyna: IFAC, 2010, s. 97-102. ISBN 978-3-902661-95-1. ISSN 1474-6670.

[x] Přikryl, Z., Husár, A., Hruška, T. a Masařík, K.: *ASIP Design in the Lissom Project*. In: ACACES 2010 - Poster Abstracts. Ghent: High Performance and Embedded Architecture and Compilation, 2010, s. 105-108. ISBN 978-90-382-1631-7.

[xi] Přikryl, Z., Křoustek, J., Hruška, T., Kolář, D., Masařík, K. a Husár, A.: *Design and Debugging of Parallel Architectures Using the ISAC Language*. In: Proceedings ot the Annual International Conference on Advanced Distributed and Parallel Computing and Real-Time and Embedded Systems. Singapore: Global Science & Technology Forum, 2010, s. 213-221. ISBN 978-981-08-7656-2.

[xii] Přikryl, Z., Masařík, K., Hruška, T. a Husár, A.: *Generated Cycle-Accurate Profiler for C Language*. In: 13th EUROMICRO Conference on Digital System Design, DSD'2010. Lille: IEEE Computer Society, 2010, s. 263-268. ISBN 978-0-7695-4171-6.

[xiii] Trmač, M., Husár, A., Hranáč J., Hruška, T. a Masařík, K.: *Instructor Selector Generation from Architecture Description*. In: 6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science. Brno: Masarykova universita, 2010, s. 167-174. ISBN 978-80-87342-10-7.

[xiv] Husár, A., Přikryl, Z., Masařík, K. a Hruška, T.: *ASIP Design using Architecture Description Language ISAC*. In: ACACES 2009 - Poster Abstracts. Ghent: High Performance and Embedded Architecture and Compilation, 2009, s. 137-139. ISBN 978-90-382-1467-2.

[xv] Přikryl, Z., Masařík, K., Hruška, T. a Husár, A.: *Fast Cycle-Accurate Interpreted Simulation*. In: Tenth International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions. Austin: IEEE Computer Society Press, 2009, s. 9-14. ISBN 978-0-7695-4000-9.