VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PROGRAMMING OF RECONFIGURABLE SYSTEMS USING A HIGHER PROGRAMMING LANGUAGE

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Ing. ADAM HUSÁR
AUTHOR

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PROGRAMOVÁNÍ REKONFIGUROVATELNÝCH SYSTÉMŮ POMOCÍ VYŠŠÍHO PROGRAMOVACÍHO JAZYKA

PROGRAMMING OF RECONFIGURABLE SYSTEMS USING A HIGHER PROGRAMMING LANGUAGE

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Ing. ADAM HUSÁR
AUTHOR

VEDOUCÍ PRÁCE prof. Ing. TOMÁŠ HRUŠKA, CSc.
SUPERVISOR

BRNO 2014

Abstrakt

Disertační práce se zaměřuje na problematiku programování a optimalizace aplikačně specifických procesorů. Cílem bylo vytvořit prostředí pro rychlou optimalizaci aplikačně specifických procesorů, které obsahuje překladač vyššího programovacího jazyka a šablony procesoru. Proces generování překladače je rozdělen na dva kroky: extrakce sémantiky a generování backendu (zadní části) překladače. Autor také navrhnul několik procesorových jader, ty jsou zde popsány. Společně s generátorem překladače a s rozšiřitelnými procesorovými jádry autor vytvořil nástroj pro rychlou optimalizaci aplikačně specifických procesorů.

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 that includes a higher level language compiler and a processor template. The process of compiler generation is divided into two steps: semantics extraction and compiler backend generation. The author also designed several extensible processor cores and they are also described here. Together with the generated compiler and the extensible processor cores, the author created a tool for fast ASIP optimization.

Klíčová slova

Překladače, architektury procesorů, procesory s aplikačně specifickou instrukční sadou, instrukční rozšíření, jazyky pro popis architektury, vestavěné systémy.

Keywords

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

Citace

Adam Husár: Programming of Reconfigurable Systems using a Higher Programming Language, disertační práce, Brno, FIT VUT v Brně, 2014

Programming of Reconfigurable Systems using a Higher Programming Language

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc.

. . . .
Adam Husár
September 30, 2014

Poděkování

Zde bych rád poděkoval svému školiteli profesoru Tomáši Hruškovi za jeho vedení a velkou podporu po celou dobu mého studia. Dále bych rád poděkoval kolegům Karlu Masaříkovi, Zdeňku Přikrylovi, Janu Hranáčovi, Luďku Dolíhalovi, Ondřeji Ilčíkovi, Liboru Vašíčkovi, Róbertu Baručákovi, Mariánu Pristachovi, Martinu Ministrovi, Josefu Potěšilovi, Tomáši Mináčovi, Pavlu Šnoblovi, Albertu Mikó, Filipu Bennovi a dalším za skvělou spolupráci, jejich rady a nápady.
V neposlední řadě také velmi děkuji svým rodičům Slavomíru a Evě Husárovým a své přítelkyni Kateřině Lazarové za jejich lásku a podporu.

Contents

Chapter 1

Introduction

1.1 Programming of Computing Systems

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

The problem of compiling for computing systems can be defined as a problem of translating from a human-readable description of some problem solution into a form that can be understood by underlying hardware that does the problem solving.

Computing systems are designed with the goal of solving some problem. A designer first describes the problem, using a natural language. The problem description is then passed on to a programmer. The programmer's task is to translate the problem into an unambiguous description in a programming language. Tools are then used that translate the programming language description into machine code in the Instruction Set Architecture (ISA) language. This process is shown in Figure 1.1.

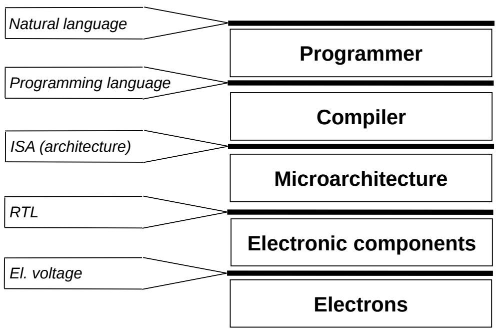


Figure 1.1: Illustrative scheme of a programmable computing system [85]

Each step of the translation simplifies the initially possibly complex problem into smaller steps. The same principle also applies to the microarchitecture. By now simple instructions are further subdivided into small steps that can be computed using just two types of basic elements: combinational logic and memory storages. The two basic elements are composed of electronic components such as transistors, capacitors, and resistors. These electronic components can finally control the flow and storage of electrons that are used as the working force when solving the original problem. In this way, the original problem statement in some natural language such as English is translated into the „language of electrons".

This illustration shows the main features of a computing system with a processor. The first feature is that the compiler and hardware are not some separate components, the two of them form the computing system. The second feature is that this illustration shows clearly the hardware/software interface which is the ISA.

1.2 Acceleration Computation with Application-Specific Instruction-Set Processors

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), 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.

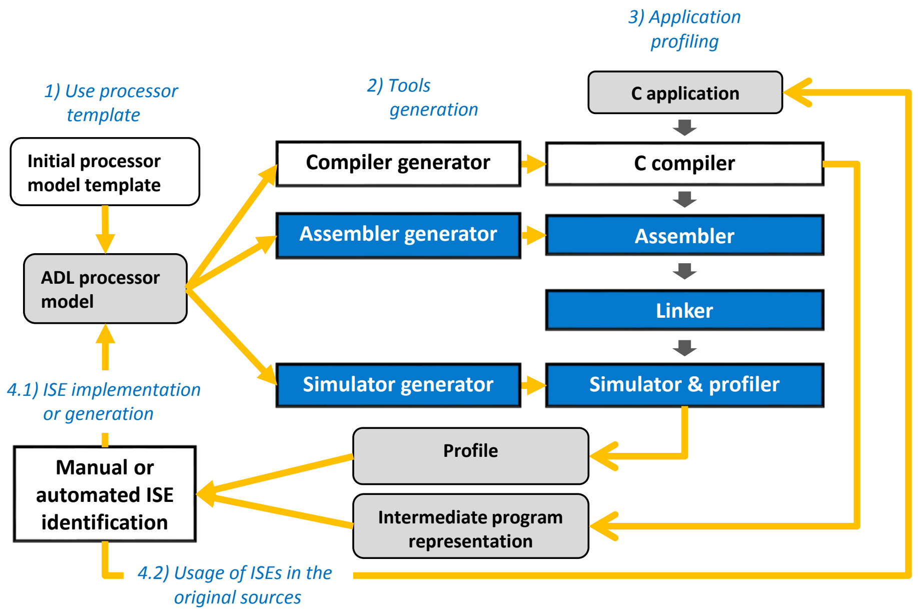
The problem is now how to efficiently design such ASIPs with ISEs.

1.3 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 [66] was started at the Faculty of Information Technology 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 was 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 [71], 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 1.2. The process consists of the following steps: 1) take a processor template in the form of an ADL model, 2) generate the compiler, assembler, and simulator, 3) compile and profile the application, 4) identify, either manually or automatically, new ISEs based on a profile (the most executed code), add the ISEs 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.



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 1.2: 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

The author started his doctoral studies in the year 2007. At that time, we already had an ADL language called ISAC, simulator generator, linker, and also assembler generator. The assembler generator was implemented by the author as his master's thesis [39].

Components that were missing at that time are shown in Figure 1.2 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.

In chapter 2, there is an overview of the current state of the art in the areas of ASIP design tools, compiler generation, extensible processor cores, ISE identification, and ISE generation. Chapter 3 then describes solutions and design of a compiler generator, extensible processor cores, and a tool for ASIP optimization made by the author.

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.

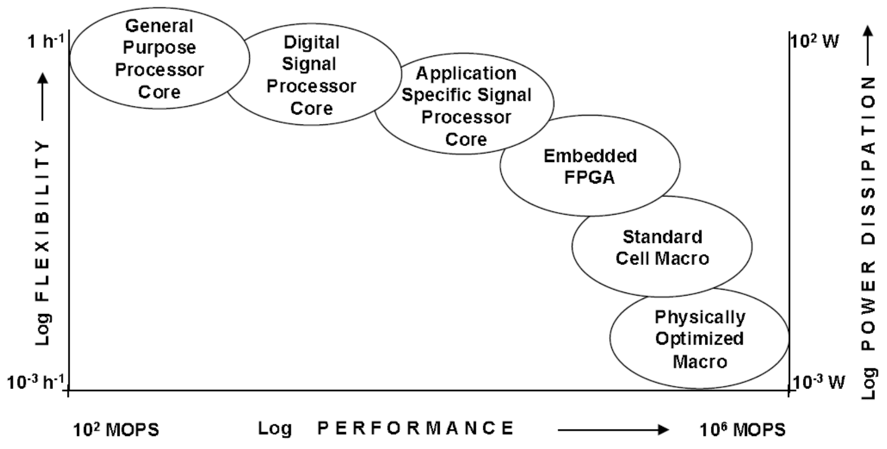
To create an embedded system that conforms to the given requirements, a correct mix of processor cores and accelerators must be chosen.

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.

The thesis focuses mainly on the application specific processor cores, which offer an ideal trade-off between flexibility and performance for some applications.

2.1.1 Application-Specific Instruction-Set Processors

Application-Specific Instruction-Set Processors (ASIPs) and their reconfigurability are the main focus of the thesis. ASIPs were already briefly described in the Introduction 1.2 and we will go into more details now. ASIPs are a mix of General Purpose Processors (GPPs)



1 h⁻¹
10⁻³ h⁻¹
Log FLEXIBILITY
General Purpose Processor Core
Digital Signal Processor Core
Application Specific Signal Processor Core
Embedded FPGA
Standard Cell Macro
Physically Optimized Macro
10² W
10⁻³ W
Log POWER DISSIPATION
10² MOPS
Log PERFORMANCE
10⁶ MOPS

Figure 2.1: Trade-off between flexibility, performance and power consumption [12]

and Application Specific Integrated Circuits (ASICs). ASIPs are fully programmable in the same way as general-purpose processors, contain special instructions and other extensions that have access to the register file and also to the memory.

The main disadvantage of ASIPs is the need to fabricate a new chip once ISEs have been designed. This can be alleviated by using reconfigurable hardware for ISE synthesis. The approaches to reconfigurable ASIP design can be divided into two categories, where one focuses on fine-grained reconfigurable arrays such as FPGAs and coarse-grained reconfigurable arrays (CGRAs) [97].

The main goal when designing an application-specific instruction set processor is to *minimize the semantic gap* or *software/hardware mismatch* between the application code in a higher-level programming language and the code in an instruction set language. The minimization of this semantic gap is usually obtained by introducing instruction set extensions (ISEs) that better match constructs from the application program than the standard instructions from the processor's instruction set. Instead of many simple instructions, just one special instruction may be executed with the same result and this usually greatly improves performance.

ASIPs are usually designed using Architecture Description Languages.

2.1.2 Architecture Description Languages

Architecture description languages (ADLs), also sometimes called processor description languages (PDLs), are modeling languages targeted at processor design.

They can be divided into three main categories based on their level of abstraction [79]: 1) *structural ADLs* (e.g. MIMOLA, MESCAL, AIDL, UDL/I, xADL) contain microarchitectural information and are aimed mainly at synthesis, validation, and precise simulation, 2) *mixed ADLs* (e.g. ISAC, CodAL, LISA, Tensilica TIE, EXPRESSION, Maril, MESCAL/-MADL, HMDES/MDES, IDL, RADL) are the most wide-spread and describe both architecture and microarchitecture and should be suitable for synthesis, validation, simulation, and compilation, and 3) *behavioral ADLs* (e.g. nML, ISDL, CSDL), which describe only the architecture and are designed for functional simulation and compilation.

These languages differ substantially in the level of detail and in their suitability for compiler generation. The more structural information is captured, the harder it usually is to extract instructions with their semantics that is usable for compiler generation. This makes the behavioral languages with functional instruction semantics the best candidates for compiler generation but it is necessary to provide a useful tool, and also an automated path to hardware generation. This is the reason why mixed languages are the most popular.

2.1.3 CodAL Language

The CodAL [20] 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 lightweight, allows fast design space exploration, and 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 [15] or by using functional verification [114]. This way the instruction-accurate model can be seen as a golden model, and the cycle-accurate model is a more detailed 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 [21]. Codasip Framework is an EDA (Electronic Design Automation) tool for fast ASIP design. Using the Eclipse-based Codasip Studio graphical interface, the user defines models in CodAL. From the processor model can be generated the C compiler, assembler, simulators with different accuracies and profiling levels with debugging capabilities, VHDL [5] description, and verification supporting tools.

Instruction Accurate CodAL Models

The instruction accurate CodAL model (IA model) captures the instruction set architecture (ISA) of a processor core. Such an architectural description is only concerned with what instructions, registers, and memories does the processor provide for the programmer. The IA model is in this way split into two parts: processor resources (registers, and memories) and instruction set definition (instructions). We will show an example of the Codix uRISC architecture created by the author as a tutorial model for the CodAL language. We first define a program counter register that must be present in every von-Neumann based processor.

```
program_counter bit[32] pc;
```

The Codix uRISC processor has 32 32-bit general purpose registers. With the `arch` keyword it is specified that these registers are architectural (accessible to the programmer). Non-architectural registers only store some intermediate results during the execution of instructions and they do not affect the architectural state of the processor.

```
arch register bit[32] regs[32];
```

We also need a memory, the Codix uRISC processor has 32-bit words, where each 8-bit byte can be addressed.

```
memory bit[32] mem {
  .endianess = big,
  // Least addressable unit - smallest addressable element
  // in memory is an 8-bit byte.
  .lau = 8,
  // Read, write, and execute flags
  .flags = {r, w, x},
  // Size in words
  .size = 0x400000
};
```

Once we have described the processor resources, we can describe the instructions. The simplest instruction is a no-operation instruction.

```
element instruction_nop
{
  // Syntax
  assembler { "NOP" };

  // Binary coding, this example shows 2 ways of binary constants
  // definitions: as a binary number (0b000000), and as decimal
  // or hexadecimal number with a specified bit width (0:26).
  binary { 0b000000 0:26 };
};
```

To define a more complex instruction, we will need register operands; they can be defined in a separate `element` like this:

```
element gpreg
{
  // Assembler section defines the syntax of registers, in this
  // case the registers are written as Rx, where x is the register
  // number.
  assembler { "R"~index=unsigned };

  // Codix uRISC processor has 32 registers, therefore we need
  // 5 bits to encode register address/index.
  binary { index=0b[5] };

  // An element may return an integer value.
  // This element returns the register index.
  // A return section may contain any C language expression
  // terminated with a semicolon.
  return { index; };
};
```

Parts of instructions or whole instructions are described by a construct `element`. To make the ASIP description shorter, it is possible to use a `set` that denotes a set of elements with a common identifier.

```
// Definition of operation code constants.
#define OPC_MOV 2
#define OPC_NEG 3

// The two following elements define operation codes and their
// mnemonics for the MOV and NEG instructions. The return section
// is used to return the operation code value.
element opc_mov
{
  assembler { "MOV" }; binary { OPC_MOV:6 }; return { 2; };
};

element opc_neg
{
  assembler { "NEG" }; binary { OPC_NEG:6 }; return { 3; };
};

// In all places where the opc_mov_neg group is used,
// either opc_mov or opc_neg may be used in its place.
set opc_mov_neg = opc_mov, opc_neg;
```

Instructions that perform register value move and negation are shown here:

```
element i_2_reg_operands
{
  // Element instances are like local variables.
  use gpreg  as  reg_dst, reg_src;

  // If only one instance of a certain operation or a group is used,
  // there is no need to assign it an unique name.
  use opc_mov_neg;

  assembler { opc_mov_neg reg_dst "," reg_src };
  binary { opc_mov_neg reg_dst reg_src 0:16  };

  // Now we describe an instruction behavior using the C language.
  semantics
  {
    // Operation's semantics is defined with the C language.
    // Register instances reg_src and reg_dst contain
    // register operand addresses. The instance opc_mov_neg
    // is the operation code.
    switch (opc_mov_neg)
    {
    case OPC_MOV:
      regs[reg_dst] = regs[reg_src];
      break;
    case OPC_NEG:
      regs[reg_dst] = ~regs[reg_src];
      break;
    }
  };
};
```

Instruction accurate CodAL models are well suited for fast design space exploration, because the user can quickly add or remove instructions, generate the C compiler, assembler, simulator, and test new instructions added to the model.

Cycle Accurate CodAL Models

Cycle accurate (CA) CodAL models define the processor's pipeline. The description can be quite complex, because pipeline stages, memory interfaces, control data and structural hazard handling are defined in the CA model. Due to this detailed description, the RTL (Register Transfer Level) hardware description can be generated. The verification environment, simulator, assembler, and disassembler can also be generated from a CA model. As an example, the CA model of the Codix uRISC architecture is shown in this section.

First, registers and memory resources are defined. Certain registers can then be assigned to a pipeline stage. All registers of one pipeline stage can be stalled (they are not overwritten on a new clock cycle) or cleared (reset), which simplifies the processor definition. The registers with names ex_ and wb_ are defined as non-architectural registers. The assignment of these registers to the pipeline stages is shown below.

```
pipeline pipe {
  FE   : ;
  ID   : ;
  EX   : ex_pipeline_mux, ex_dest_we_mux, ex_pc_we_mux, ex_mem_op,
         ex_alu_op, ex_addrW, ex_condition, ex_writedata,
         ex_operand_B, ex_operand_A, ex_opcode;
  WB   : wb_pipeline_mux, wb_addrW, wb_dest_we, wb_mem_op, wb_result,
         wb_opcode;
};
```

An instruction syntax and binary coding can be defined by `elements` and `sets` in the same way as in an IA model. The syntax and coding information can be shared between the IA and CA models.

Then there are `events`. Events model the mostly combinational logic in pipeline stages. A simple event in the FE (fetch) stage that issues a request to the instruction memory while expecting that the request will be successful (no stall, etc.) is shown here:

```
signal bit[32] instruction_address;
signal bit[1] ex_pc_we;

event fe : pipe.FE {
  use pipeline_control;

  semantics {
    // If a jump is being executed (signal ex_pc_we is set to 1),
    // use the jump target address, otherwise use the value
    // from the register pc.
    instruction_address = (ex_pc_we) ? ex_result : pc;

    // In Cycle Accurate models, every memory access is synchronous.
    // In the first cycle is a memory request read issued and in the
    // following cycle the value can be read.
    mem_fetch.request(CP_RQ_READ, instruction_address);

    // We will use the next instruction address in the next cycle.
    pc = instruction_address + 4;
  };
};
```

In the ID (instruction decode) stage the instruction is read from the memory and sent to the instruction decoder.

```
event id : pipe.ID {
  use instr_asm;
  use instr_hw;
  use id_output;

  // Definition of the instruction decoder for the hardware and
  // simulator. HW decoder decodes the instruction and sets
  // the pipeline control signals.
  decoders {
    { instr_hw(id_opcode); }
  };
  // Definition of the instruction decoder for assembler
  // and disassembler. The instance instr_asm defines the syntax
  // and binary coding of all instructions using sets and elements.
  // HW decoder can also be generated from the element instr_asm,
  // but it will have a 32-bit input and will be more complex than
  // a decoder generated from the 6-bit decoder specification that
  // uses the instance instr_hw.
  start {
    { instr_asm; }
  };

  // Additional C code that prepares instruction opcode value
  // for the HW decoder and extracts additional operands from the
  // 32-bit instruction based on the processor binary instruction
  // formats.
  semantics {
    uint32 id_instruction;

    // We read the instruction requested from the instruction
    // memory in the previous stage fetch.
    mem_fetch.ifinish(CP_IF_READ, id_instruction);

    // Opcode passed to the instruction decoder.
    id_opcode = (id_instruction >> 26) & 0x3F;

    // Destination, and source register addresses,
    id_addrW = (id_instruction >> 21) & 0x1F;
    id_addrA = (id_instruction >> 16) & 0x1F;
    id_addrB = (id_instruction >> 11) & 0x1F;

    // Immediate operands.
    id_imm26 = id_instruction & 0x03FFFFFF;
    id_imm16 = id_instruction & 0xFFFF;
  };

  timing {
    // Perform other operations in the IF stage such as preparation
    // of values for interstage registers ex_operand_A
    // and ex_operand_B.
    id_output;
  };
};
```

And then there is the EX (execute) stage containing an ALU that performs operations specified by the instruction being executed.

```
event ex : pipe.EX {
  semantics {
    // Register ex_alu_op was set by the instruction decoder based
    // on the 6-bit instruction opcode. Operands were either read
    // from the register file, or an immediate operand is used.
    switch (ex_alu_op) {
    // Move instructions
    case EX_MOV:
      ex_result = ex_operand_A; break;
    case EX_MOVSI:
      ex_result = ex_operand_B; break;

    // Arithmetic instructions
    case EX_ADD:
      ex_result = ex_operand_A + ex_operand_B; break;
    case EX_SUB:
      ex_result = ex_operand_A - ex_operand_B; break;
    case EX_MUL:
      ex_result =
        (uint32) ex_operand_A * ex_operand_B; break;
    ...
      }
  };
};
```

Such a description of the CA CodAL model can be processed and, using high-level synthesis approaches, synthesized for FPGAs or ASICs. We described the IA and CA abstraction levels of the CodAL language and in the next section we will describe how the CodAL language can be used for the design and optimization of a processor core.

The two components needed in the ASIP optimization flow as shown in Figure 1.2 are the C compiler and C compiler generator. In the following section, we will review the problematics of retargetable compilation.

2.2 Retargetable Compilers

In this section, we will review retargetable compilers and the information they need about the target architecture. Special focus will be put on LLVM, because LLVM retargeting is one of the partial goals of the thesis.

2.2.1 Higher Language Level Compiler Structure

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

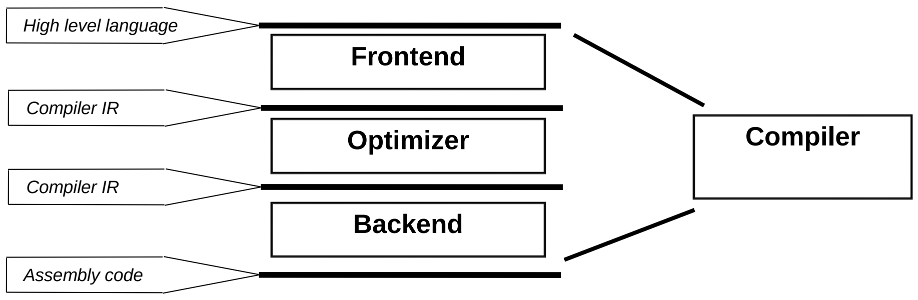


Figure 2.2: 3-part HLL compiler structure

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.

2.2.2 Retargetable Compilers

A retargetable compiler can be classified either as [62]:

- *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.

The existing *parametrizable* compilers always target the same base architecture and allow choosing between different subtargets (e.g. choosing a specific version of the ISA and enabling some special extensions and sets of instructions such as floating-point instructions).

User-retargetable compilers are those generated from an ADL description; the compiler generator is able to analyze the ADL model and transform it with minimal user interference into a description for a developer-retargetable compiler. The class of *developer-retargetable* compilers is the most common class. The GCC and LLVM compilers belong to this category.

A higher-level definition of the instruction set definition is usually available, but a lot of coding in C or C++ must be done to define the remaining architecture particularities. To write this code, detailed knowledge of the compiler platform is necessary.

Examples of retargetable compilers are gcc, LLVM, CoSy, SUIF, lcc, Trimaran, LANCE, and SPAM [62]. Target features may vary substantially and no simple but powerful enough architecture description has been designed yet, and so all these listed compilers require extensive compiler expertise to retarget a compiler.

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.

In the following sections, we will analyze the most popular retargetable compilers and show how they can be retargeted.

2.2.3 LLVM

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based [101] compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages [59]. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects.

All the compilation steps in the LLVM compilation framework are shown in Figure 2.3. The input program is first parsed by a frontend and then it is optimized. Now comes a step that differentiates LLVM from other compilers: it is its ability to link modules on the intermediate representation level (the .bc file suffix represents files in LLVM intermediate representation). Using the llvm-link, the already compiled and optimized modules are merged together and the following optimizer can do some additional interprocedural optimizations such as inlining. This process, where modules in an intermediate representation are put together and then optimized is called whole program optimization (WPO), misleadingly also called link-time optimization (LTO). The resulting program representation is passed either to the target code generator (backend), which generates the assembly code, or to the LLVM virtual machine. In this text, we will focus mainly on the native execution path and the target code generator.

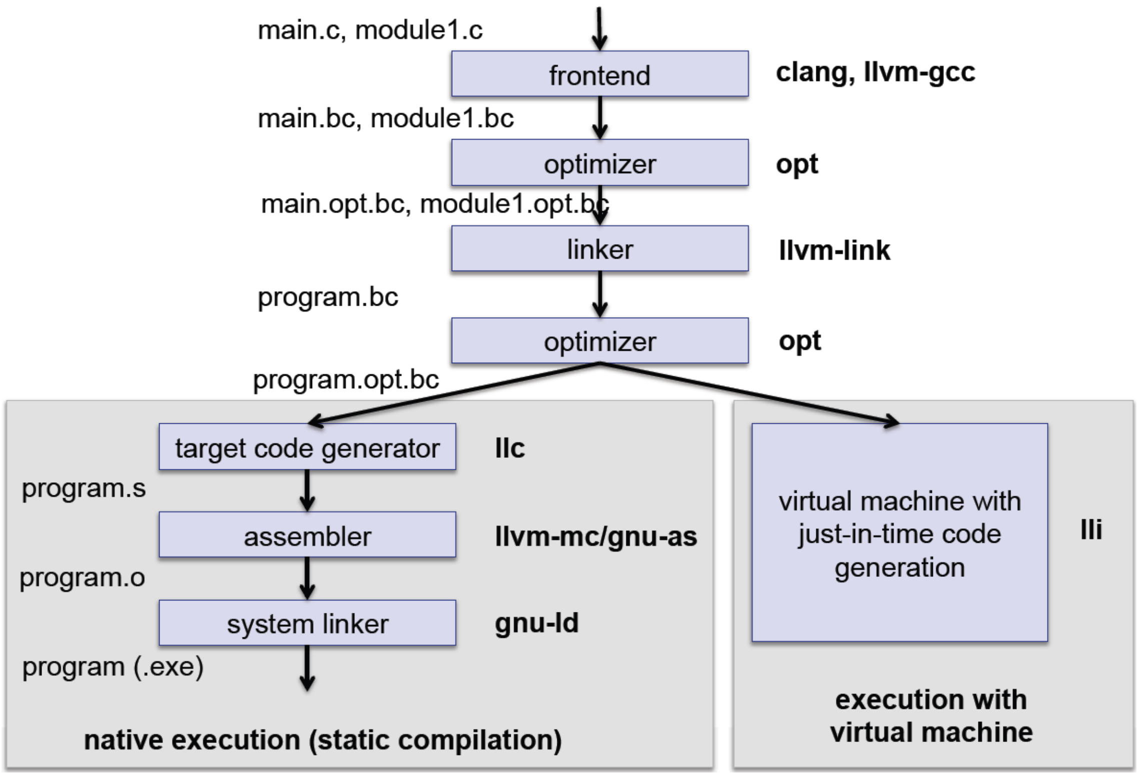


Figure 2.3: LLVM compilation framework overview [86]

We will describe the most important steps of compilation with the LLVM framework: the frontend clang, intermediate optimization, and with the main focus on the retargetable compiler backend.

Clang Frontend

The input source code is parsed with a frontend *clang* that transforms the input code into an Abstract Syntax Tree (AST). Semantic checks are performed over this AST. There is also a static analyzer *clang-analyzer* tool that works on this AST and can statically detect bugs in a program. The AST representation can be used to generate a code almost identical to the code that was originally parsed, so that *clang* can be used to do source-to-source transformations [57]. After semantic checks, a program in the LLVM IR compiler internal representation is generated.

The frontend for the C and C++ languages needs to know the target architecture type sizes and alignment and this is specified by the target architecture triple. The target triple is in the form of an architecture-vendor-operating system. In fact, only the architecture specification controls the frontend behavior. *Clang* contains a table of supported architectures that contain data type sizes, alignment, and endianness. For example, for the 32-bit little-endian ARM [4] architecture it knows that integer variables and pointers are 32 bits wide. For the 16-bit MSP430 [50] architecture it knows that integers and pointers are 16 bits wide, and the memory is organized in the little endian manner.

Before *clang* was available, a modified GCC frontend *llvm-gcc* was used to generate LLVM IR. As most applications were originally compiled with GCC, *clang* tends to have a fully compatible command-line interface.

The *clang* frontend is also used as a library in graphical IDEs to parse and analyze source codes opened in a graphical editor. One interesting feature of *clang* is that it parses the C or C++ language input together with the C language preprocessor directives. Thus in a graphical editor, *clang* does not need to preprocess the whole source file, it can parse, for example, only one function to check syntactic and semantic errors and display them to the user.

LLVM IR

The LLVM Intermediate Representation (LLVM IR) is a Static Single Assignment (SSA) (e.g. [101]) based representation that provides type safety, low-level operations, flexibility, and the capability of representing high-level languages clearly. It is a common code representation used throughout all the phases of the LLVM compilation strategy.

LLVM IR contains standard integer and floating point arithmetical and logical operations, conversions between different data types, comparisons, memory access operations and address computations, memory synchronization, and control flow instructions. There are also special operations for garbage collector, exception handling, and also for standard C library operations such as memcpy, or memmove. Some debugging information directives are also present in the IR code as instructions.

The high-level constructs, e.g. for exception handling or switch instruction, cannot be mapped directly to any hardware instructions, so the LLVM IR architecture is in fact virtual and its level of abstraction is quite similar to the representations of the Java Dalvik [22] for Java or Microsoft Common Language Interface [53] for C#. LLVM IR is also a register-based representation unlike the Oracle Java bytecode [65], which is stack-based.

One big difference to the Java and C# intermediate representations mentioned is that LLVM IR supports integers of arbitrary bitwidth. This allows using it not only for programming language compilation but also, for instance, for hardware synthesis, which needs to express exact signal and register bitwidths [120].

One interesting feature of the LLVM IR virtual architecture is that it contains an unlimited number of virtual registers. It is a consequence of using the Static Single Assignment form.

Static Single Assignment Form

The Static Single Assignment (SSA) [24] form specifies that one variable must be assigned only once. SSA facilitates the implementation of numerous analyses and transformations. As one example, the use-def chain analysis [81], which says where a variable was defined and where this definition is used, is very straightforward.

But what happens in cases when we need to assign a register several times, e.g. in a loop or in branches as in the following example [86].

```
int max_square(int x, int y)
{
    int result = 0;
    if (x>y)
        result = x*x;
    else
        result = y*y;

    return result;
}
```

The non-SSA control flow graph of this function is as shown in Figure 2.4.

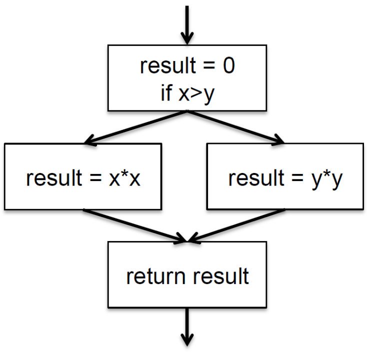


Figure 2.4: Non-SSA control flow graph

A special instruction must be present in any SSA-based IR and is usually called the *phi-node* or the *phi instruction* ϕ. Its result depends on which control flow path was taken and it uses the corresponding value (like a select instruction or a multiplexer). The same program, but now in the SSA form with the phi-node instruction, is shown in Figure 2.5.

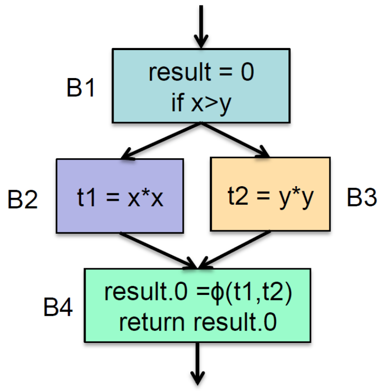


Figure 2.5: SSA control flow graph

If during execution the control flow goes through the phi instruction in the basic block B4, the variable result.0 is set to t1, when the previously executed basic block was B2, but to t2, when the previous basic block was B3.

The example code is shown in Figure 2.6 in the LLVM IR textual form, where you can also see the phi instruction.

There are efficient translation schemes of the *phi* instruction for processor architectures (e.g. [101], [16]), so there was no need to have such an instruction in instruction sets

```
define i32 @max_square(i32 %x, i32 %y) {
  %1 = icmp sgt i32 %x, %y
  br i1 %1, label %2, label %4

; <label>:2
  %3 = mul nsw i32 %x, %x
  br label %6

; <label>:4
  %5 = mul nsw i32 %y, %y
  br label %6

; <label>:6
  %result.0 = phi i32 [ %3, %2 ], [ %5, %4 ]
  ret i32 %result.0
}
```

Figure 2.6: LLVM IR code with phi instruction

with hardware implementation. In our example, it suffices that the variables t1 and t2 are mapped to the same physical register, then the phi instruction can be ignored.

Besides the static single assignment form, also a dynamic single assignment of intermediate program representation form is also used, currently mainly in the research into parallel compilation, e.g. [111].

Optimizer

The LLVM optimizer is a set of analysis and transformation passes executed by the so-called pass manager. Each pass can specify its analysis dependencies and then the pass manager should schedule the computations of analyses before the pass that needs them is executed. The dependency computation does not work perfectly in practice, but it is still very easy to add a new pass or to execute just a particular transformation in the optimizer.

The LLVM optimizer works only with LLVM IR. As regards retargetability, the only thing the optimizer needs to know about the target architecture is called *data layout* [68], with one small exception.

The only small exception noted above is the autovectorization passes that need to know the SIMD-related capabilities of the target architecture to perform efficient vectorization transformations.

An example of the data layout specification is shown here:

```
E-p:32:32:32-S64-n32-i32:32:32-f32:32:32-i64:32:32-f64:32:32
```

The first 'E' specifies endianness. The p:32:32:32 says that a pointer is 32 bits wide, according to the by ABI (Application Binary Interface, e.g. [95]) specification a pointer must be aligned on 32 bits, and the the preferred alignment is also 32 bits. The S64 specifies that alignment on a stack is 64 bits, n32 says that native integers are 32 bits wide, and the following triples specify ABI and the preferred alignment for specific data types such as int32.

The data layout is a numerical architecture description and this puts the LLVM optimizer into the category of parametrizable compilers as it targets only the LLVM IR architecture.

Backend Overview

The LLVM backend takes LLVM IR as the input and produces either a textual assembly code or directly an object file. We will focus only on the textual assembly code generation. Direct object file generation is useful only for improving the compilation performance and from the viewpoint of high-level programming compilation language; it is not different from assembly code generation. The main transformations that are target-dependent in the LLVM backend are:

1. lowering,
2. legalization,
3. instruction selection,
4. register allocation,
5. prologue/epilogue insertion and frame finalization,
6. scheduling, and
7. assembly printing.

In the rest of this subsection, we will explain in detail the purpose of these passes and how they are retargeted for different architectures. For each pass, we will show the input architecture description.

Also, a running example will be used to show what kinds of transformations are made in each of these passes. As an example, we will use a simple function that adds two 64-bit integers and stores them into a global variable. However, the target architecture has instructions that work only over 32-bit integers, so the code needs to be transformed into an equivalent sequence of operations supported by the target architecture.

This is the original C code:

```c
long long x;

void ladd(long long a, long long b)
{
    x = a + b;
}
```

After compilation with the frontend and optimizer, the resulting LLVM IR is as follows:

```
define void @ladd(i64 %a, i64 %b) #0 {
entry:
  %add = add nsw i64 %b, %a
  store i64 %add, i64* @x, align 8, !tbaa !1
  ret void
}
```

For the examples of the architecture definition, we will use a simple architecture Codix uRISC whose instruction set is minimal with regards to the needs of the LLVM backend infrastructure. Codix uRISC is described in Appendix B.

Lowering

Lowering is the first target-dependent transformation in the LLVM backend. The main task of the lowering pass is to transform the input LLVM IR into an instruction selection DAG (directed acyclic graph) (e.g. [2]. Lowering needs information on architecture register files and on the calling convention, so that it can prepare the code for the retrieval of function arguments such as the variables `a` and `b` in our example.

Registers Definition There are generally two types of registers: physical and logical. The use of a physical or a hardware register by an instruction is defined by its operation code (opcode; a classical example of the physical register is the carry flag in an addition instruction or a fixed return address register in the function call instruction (e.g. when a CALL instruction always stores the return address into the register R31). Logical registers on the other hand are addressed with an address stored in the instruction binary coding. Logical registers are usually used as instruction register operands, for example where an addition instruction can use any general purpose registers as inputs and output. All logical registers can be accessed as physical registers.

In LLVM, the registers are defined in the file `RegisterInfo.td`. LLVM uses a special language for architecture definition, called *tablegen* [69]. Definition files in this language using the extension `.td` are transformed into various C++ source files during backend compilation.

First, all physical registers are defined. For each physical register its assembly syntax is defined (if available), and a DWARF index for debugging information is assigned.

The following examples use the *tablegen* tool input language.

```
// "R0" is assembly syntax, gpreg_ani is alternative name index,
// and DwarfRegNum specifies debugging information mapping index.
def regs_0: URiscReg<["R0"], [gpreg_ani]>, DwarfRegNum<[0]>;
def regs_1: URiscReg<["R1"], [gpreg_ani]>, DwarfRegNum<[1]>;
...
def regs_31: URiscReg<["R31"], [gpreg_ani]>, DwarfRegNum<[31]>;
```

Register definitions are then used to define classes of physical registers. These classes have assigned data types that these registers can hold. The class `regs` is a class that contains 32-bit physical registers able to hold 32-bit integers (i32). When an instruction such as CALL uses a physical register R31, it references this physical register class `regs`.

```
// Definition of a physical register class for implicit register
// operands.
def regs: RegisterClass<"URisc", [i32]
  (add regs_0, regs_1, ..., regs_31)>
{
  // Size of a register value when stored in the memory.
  let Size = 32;
}
```

The logical register class is defined like the physical register class. Registers from the logical register class are used as instruction operands, where any of the logical registers is allowed. The class `gpreg` is then referenced in the definition of an arithmetic instructions such as ADD.

```
// Definition of a logical register class for explicit register
// operands.
def gpreg: RegisterClass<"URisc", [i32], 32,
  (add regs_0, regs_1, ..., regs_31), gpreg_ani>
{
  // Order of register use by the register allocator
  let AltOrders = [(add regs_2, regs_3, ..., regs_29)];
  let Size = 32;
}
```

The usage of physical registers is given by the ABI register usage specified in the calling convention and by particular instructions that were used or selected, whereas the usage of logical registers is determined by register allocator.

Calling Convention The second source of target information for lowering is the calling convention. This is in fact the main part of the ABI definition and specifies how parameters must be passed to a function and how the result values are returned. The calling convention is defined for LLVM in the `CallingConv.td` file.

Function argument passing is defined by the `CC_URisc_Gen` calling convention, where the registers `regs_2 - regs_5` are used to pass 32-bit integer (i32) function arguments. Wider values such as i64 or structures are split into 32-bit chunks. If a structure wider than 128 bits is passed, it does not fit into the 4 x 32 bits and it is stored on the stack as a whole.

```
def CC_URisc_Gen: CallingConv<[
  CCIfType<[i16,i8,i1], CCPromoteToType<i32>>,
  CCIfType<[i32], CCAssignToReg<[regs_2, regs_3, regs_4, regs_5]>>,
  CCIfType<[i32], CCAssignToStack<4,4>>
]>;
```

Returning a value from a function is defined by `RetCC_URisc_Gen`. If a function returns a 32-bit value, then R2 (regs_2) is used. Value returning works the same was as parameter passing.

```
def RetCC_URisc_Gen: CallingConv<[
  CCIfType<[i16,i8,i1], CCPromoteToType<i32>>,
  CCIfType<[i32], CCAssignToReg<[regs_2, regs_3, regs_4, regs_5]>>
]>;
```

Code Transformation with Lowering We described the register and calling convention definitions and we can finally show and explain the result of the lowering pass in Figure 2.7.

The graph shows another internal code representation used in LLVM besides AST in the frontend and LLVM IR in the optimizer. This representation is called Selection DAG and, as its name suggests, it is used mainly for instruction selection. But before an instruction selection can be made, it needs to be transformed to better match the target architecture. The lowering pass creates this Selection DAG representation and uses the described target-specific information. For each basic block from LLVM IR, one Selection DAG is created.

The nodes `EntryToken` and `GraphRoot` mark the entry and exit nodes, respectively. The solid lines mark data dependencies while the dashed lines represent control and memory dependencies. At the top of the Figure 2.7, there are 4 virtual input registers `%vreg0-%vreg4`. Based on the available register classes and on the calling convention, the lowering pass decides that function input parameters will be passed in four 32-bit registers. These input registers are then used by an auxiliary operation `CopyFromReg`,which is a simple register copy that is usually optimized away. By means of `build_pair` two 32-bit values are merged into a 64-bit value. Resulting 64-bit values serve as input for the 64-bit addition from our example code. The addition result is then stored to a global variable, specified by `GlobalAddress`. The `store` is the last operation with a data dependency in this basic block and must be followed by a function return. Each particular backend in LLVM replaces the return operation with its own pseudo instruction that will be transformed later, in our case the return is replaced with `URiscISD::Ret`.

The lowering pass performs some target-dependent transformations, mainly with respect to function calls and argument passing, but most of the target-dependent code transformations are made in the following legalization pass.

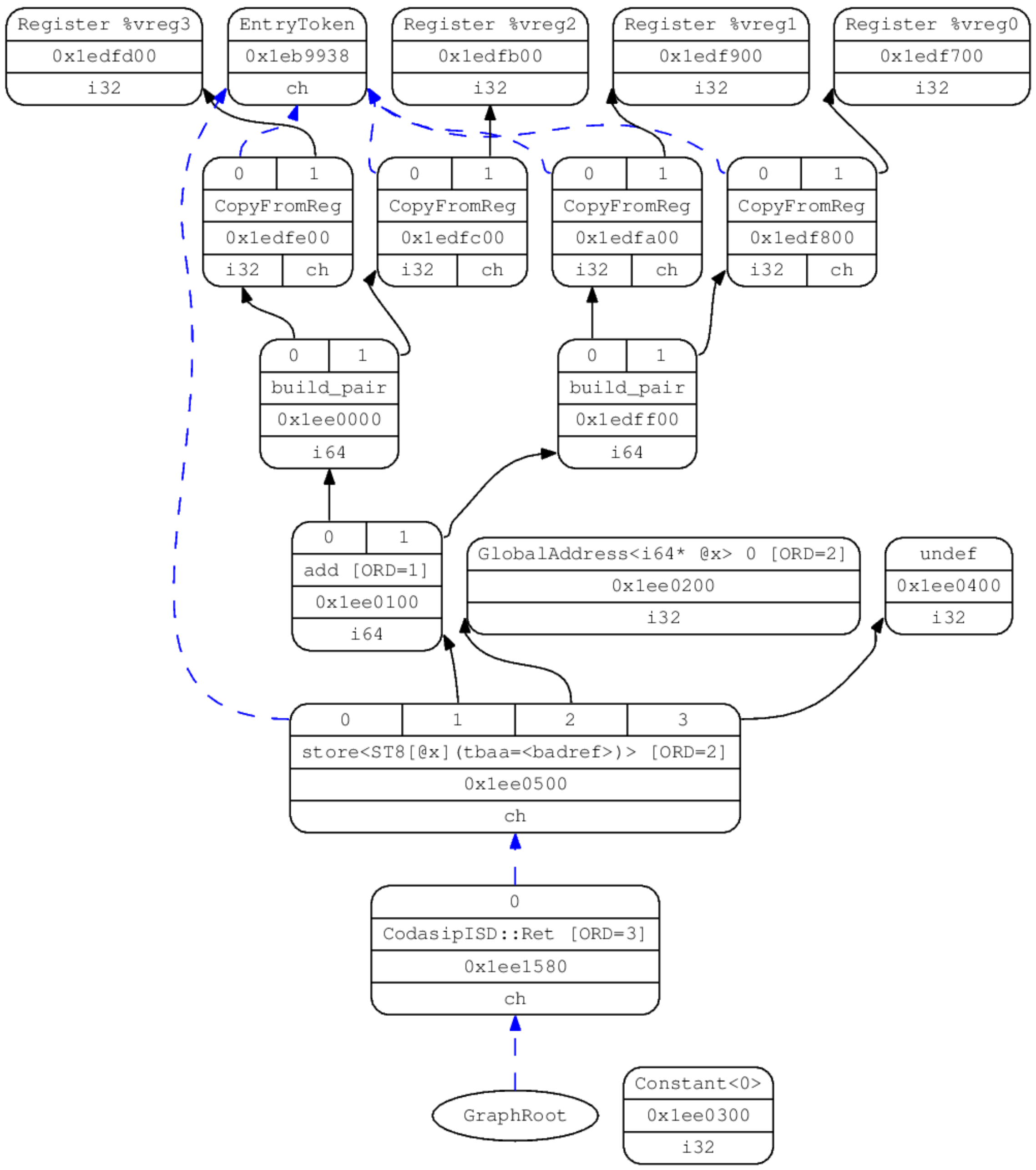


Figure 2.7: Example Selection DAG created by the lowering pass

Legalization

The main task of legalization is to replace all operations that are not supported by the target architecture either with an equivalent sequence of supported operations or with a call to a runtime library function.

Legalization in LLVM can be divided into two main parts: datatype and operation legalizations.

The goal of datatype legalization is to transform operations on unsupported data types into the same operations on supported data types. Datatype legalization in LLVM is very strong and automated, and almost any operation, for example with 64-bit operands, can be transformed into an operation with 32- or 16-bit operands.

Operation legalization then transforms unsupported operations into different supported operations. Operation legalization has low automatic support in LLVM and must be usually supplied by the user with instruction selection patterns. The main problems usually encountered with operation legalization are condition code calculation (e.g. for conditional jumps), and long immediate loading.

Legalization Specification The backend designer must specify which operations are legal in a `ISelLowering.cpp` file.

For datatype legalization it is sufficient to specify that the target provides logical register classes supporting certain data types such as i32 in our case.

For operation legalization it must be specified for each data type and operation whether this combination is *legal* or must be automatically *expanded* or whether the designer supplies the *custom* code to do the legalization manually.

An example of legalization specification is shown below (this example is in the C++ language):

```
URiscGenTargetLowering::URiscGenTargetLowering
    (URiscTargetMachine &TM)
{
  // Here it is specified that i32 is a legal data type
  addRegisterClass(MVT::i32, &URisc::gpregRegClass);

  // Only 32-bit addition is legal for our architecture,
  // additions over another data types must be automatically
  // expanded.
  setOperationAction(ISD::ADD,MVT::i32,Legal);
  setOperationAction(ISD::ADD,MVT::Other,Expand);
  ...
  // Signed division is not supported natively and must be expanded
  // if no suitable emulation is found automatically then a
  // library call is created instead.
  setOperationAction(ISD::SDIV, MVT::i32, Expand);
  setOperationAction(ISD::SDIV, MVT::Other, Expand);
  ...
  // For global address computation the custom legalization
  // code will be used.
  setOperationAction(ISD::GlobalAddress, MVT::i32, Custom);
  ...
}
```

Operations over unsupported data types will be transformed into other supported operations.

First, we will focus on operations over supported data types. A 32-bit addition will be kept as it is in the selection DAG representation. For a 32-bit signed division (SDIV) the legalizer will look what operations can be used to implement the operation. For example, if 16-bit division were available, it would transform the 32-bit division into a sequence of operations using the 16-bit division. In this case, it will not find any suitable replacement and will generate a call to the compiler runtime library.

Compiler Runtime Library LLVM provides a library called Compiler-RT [67]. The library also contains other support routines, but the main part is *builtins*. *Builtins* is a simple library that provides an implementation of the low-level target-specific hooks required by code generation and other runtime components. As an example, a full set of floating operations is implemented in the library, using elementary integer operations so that architectures without a floating point support can emulate these operations with software. The *builtins* library provides optimized implementations of this and other low-level routines, either in the target-independent C form or as a heavily-optimized assembly.

Code Transformation with Lowering In our running example, we have a 64-bit addition operation. 64-bit integers are not legal and there is no instruction either that could perform this operation. The addition will be transformed into a sequence of other supported operations.

Our example architecture does not have a carry flag that would be useful here, so LLVM decides to transform the original computation (where x_h are the higher 32 bits of a 64-bit variable x, and x_l are the lower 32 bits):

$$c_h : c_l = a_h : a_L + b_h : b_l$$

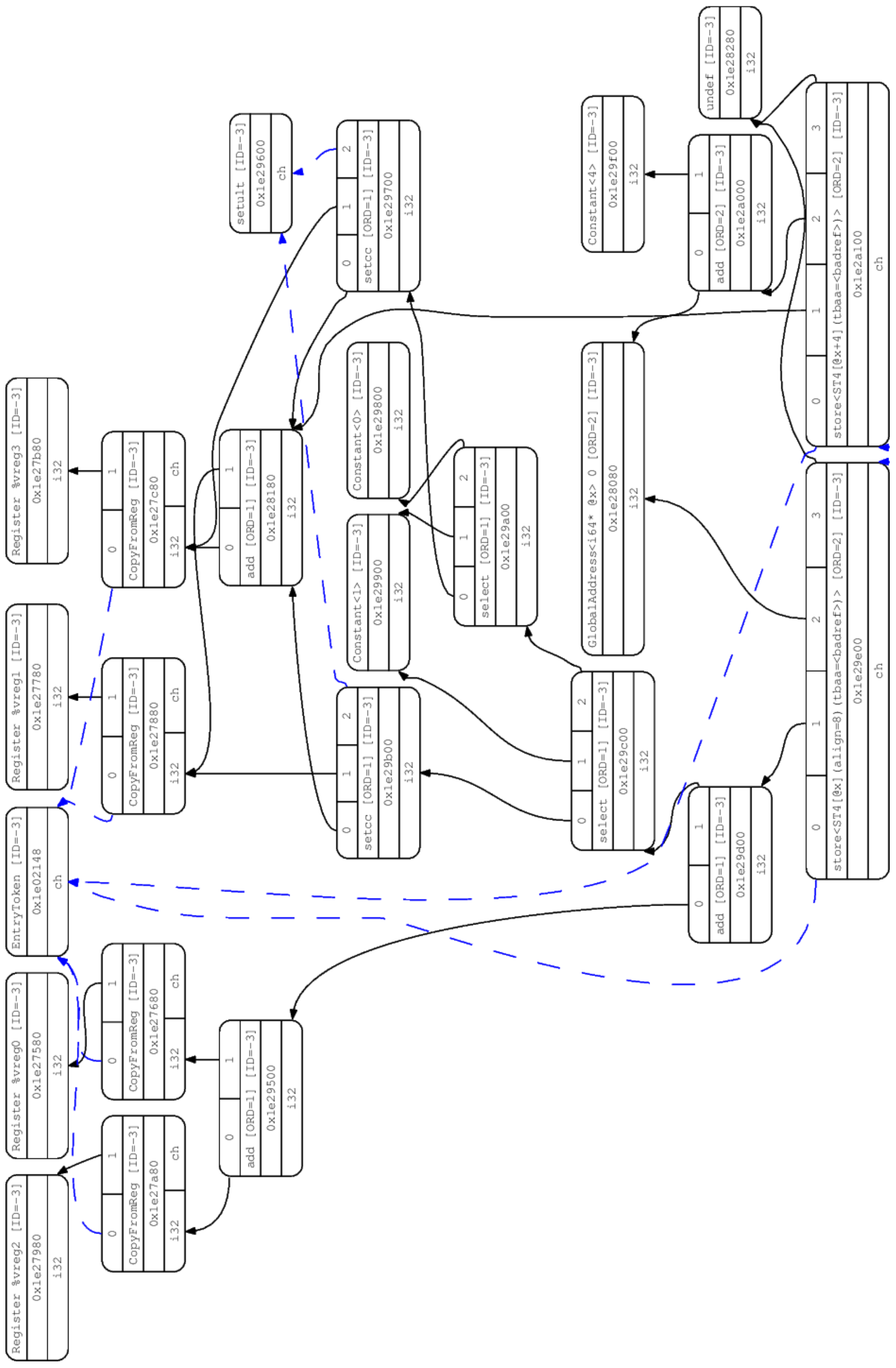
into the following sequence of computations (ternary operator ?: from the C language is used in the description):

$$c_l = a_l + b_l$$
$$carry = (c_l <= b_l)?1 : ((c_l <= b_l)?1 : 0)$$
$$c_h = a_h + b_h + carry$$

The 64-bit memory store of the computed value is also transformed into two 32-bit stores. For every legal type specified by a call to `addRegisterClass`, operations must exist that load and store the value, so the information whether a 32-bit store is available is deduced from the `addRegisterClass` call and is not specified explicitly in the legalization specification.

You can see the transformation when you compare Figures 2.7 and 2.8. The input value a_h is the register `%vreg2`, a_l is `%vreg3`, b_h is `%vreg0`, and b_l is `%vreg1`.

The goal of the legalization pass is to split complex operations into supported ones so that the instruction selector can successfully replace all target-independent Selection DAG operations with target architecture instructions.



setult [ID=-3] 0x1e29600 ch
setcc [ORD=1] [ID=-3] 0x1e29700 i32
Register %vreg3 [ID=-3] 0x1e27b80 i32
CopyFromReg [ID=-3] 0x1e27c80 i32 ch
add [ORD=1] [ID=-3] 0x1e28180 i32
Constant<0> [ID=-3] 0x1e29800 i32
Constant<4> [ID=-3] 0x1e29f00 i32
undef [ID=-3] 0x1e28280 i32
add [ORD=2] [ID=-3] 0x1e2a000 i32
store<ST4[@x+4](tbaa=<badref>)> [ORD=2] [ID=-3] 0x1e2a100 ch
GlobalAddress<i64* @x> 0 [ORD=2] [ID=-3] 0x1e28080 i32
select [ORD=1] [ID=-3] 0x1e29a00 i32
Constant<1> [ID=-3] 0x1e29900 i32
Register %vreg1 [ID=-3] 0x1e27780 i32
CopyFromReg [ID=-3] 0x1e27880 i32 ch
setcc [ORD=1] [ID=-3] 0x1e29b00 i32
select [ORD=1] [ID=-3] 0x1e29c00 i32
store<ST4[@x](align=8)(tbaa=<badref>)> [ORD=2] [ID=-3] 0x1e29e00 ch
add [ORD=1] [ID=-3] 0x1e29d00 i32
EntryToken [ID=-3] 0x1e02148 ch
Register %vreg0 [ID=-3] 0x1e27580 i32
CopyFromReg [ID=-3] 0x1e27680 i32 ch
add [ORD=1] [ID=-3] 0x1e29500 i32
Register %vreg2 [ID=-3] 0x1e27980 i32
CopyFromReg [ID=-3] 0x1e27a80 i32 ch

Figure 2.8: Selection DAG created by the legalization pass from the DAG in Figure 2.7

Instruction Selection

Instruction selection usually works on the basic block level with the basic block Selection DAG representation such as the one shown in Figure 2.8. The goal of the instruction selector is to transform this DAG with mostly LLVM-based operations as nodes into another DAG containing target architecture instructions as nodes.

All data dependencies in the input selection DAG are modeled through virtual registers and through memory dependencies. Also in the DAG after selection are most dependencies modeled still with virtual registers, but some instructions may introduce dependencies through physical registers. There is also a dependency called *glue* that does not express a physical dependency explicitly, but rather says that these instructions must be always scheduled after each other so that no other instruction in between them can, for example, rewrite flag registers set by the first instruction. Machine instructions as nodes and these dependencies then form the DAG after instruction selection.

The LLVM instruction selection pass also completely ignores register operands of instructions that are being selected; matching looks only at the LLVM SD Node operations and their data types. For example, if the target has two 32-bit integer register classes, the LLVM instruction selector combines the usage of instructions that work with the first and the second register class. It is then up to the register allocator to insert moves from one register class to another to make the code correct.

If an operation from the source DFG cannot be transformed without creating multiple basic blocks, a pseudo instruction is created. This pseudo instruction is then later transformed in a subsequent pass. This can be the case of a select operation (ternary operator in C) that, when no selects or conditional moves are available, must be transformed into an if-else construction, which requires at least two new basic blocks.

We will now review the definition of instructions for the selector.

Selection Patterns All instructions for the instruction selector (ISEL) are defined in the file `InstrInfo.td`. Similar to the calling convention specification, the *tablegen* language is used [69]. This is an example of an instruction that takes two input register operands, adds them and stores the result in another register. The `Pattern` specification is important for the instruction selector. The `add` operation is an identifier of an SD Node type and is used for matching. You can notice that the pattern is defined as a tree with the `set` operation as root.

The operand definition (e.g. `gpreg:$op0`) in the pattern is used by the register allocator to allocate registers from a correct logical register class. `AsmString` is used for assembly printing.

```
def i_3_reg_operands__opc_add__gpreg__gpreg__gpreg__:
  URiscInst <(outs gpreg:$op0), (ins gpreg:$op1, gpreg:$op2)> {
// Syntax
let AsmString = "ADD $op0, $op1, $op2";
// Pattern for instruction selector
let Pattern = [
  (set gpreg:$op0,
    (i32 (add
           (i32 gpreg:$op2),
           (i32 gpreg:$op1)
    ))
)];}
```

Such patterns can be used to transform *one-to-one* and *many-to-one* relations between SD Nodes and machine instructions. Relation *one-to-one* is the case where a target instruction perfectly matches the SD Node, *many-to-one* is when a target instruction can cover multiple SD Nodes. There are also cases when multiple instructions are needed for one node: *one-to-many* or *many-to-many* relations. In this case, the selector needs additional information on such equivalencies. They are defined in the file `Patterns.td`. The following example specifies that the move of a 32-bit global address (`tglobaladdr`) into a register can be made with the instruction `lui` followed by `ori`. The `HI_G` and `LO_G` are functions that take the higher and the lower 16 bits of an address, respectively. The instructions `i_ori__gpreg__gpreg__uimm16__` and `i_lui__gpreg__uimm16__` are defined in the file `InstrInfo.td` the same way as the addition instruction shown before.

```
def emulation_load_PTR_GA: Pattern<(i32 (GAWrap (i32 tglobaladdr:
    $op0))),
   [(i_ori__gpreg__gpreg__uimm16__
     (i_lui__gpreg__uimm16__
       (HI_G (i32 tglobaladdr:$op0))),
       (LO_G (i32 tglobaladdr:$op0))
     )]
>;
```

All these instruction definitions and patterns are processed during compiler backend compilation by the `tablegen` tool, which creates an instruction selection automaton code. This automaton then parses the input DAG produced by legalization and outputs another DAG with machine instructions.

The transformation of the addition operation made by ISEL is quite straightforward, so we will show the usage of the global address pattern. Figure 2.9 shows the global address SD Node prior to the ISEL pass. It is a part of the graph produced by legalization shown as whole in Figure 2.8.

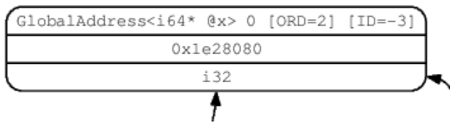


Figure 2.9: SD Node GlobalAddress that produces a 32-bit global address before the ISEL pass

Using the `emulation_load_PTR_GA` pattern, the SD Node is transformed by ISEL into the multiple instructions in Figure 2.10.

After ISEL the resulting DAG is scheduled into a list of machine instructions and the subsequent passes are performed. This DAG-to-list scheduling is very simple and only serves the purpose of linearizing the DAG, a more sophisticated pre-register allocator scheduling is performed optionally after this simple scheduling.

Register Allocation

Register allocation is a pass that assigns physical registers to originally virtual registers. The allocator in LLVM works at a function level. It first computes the liveness [81] of virtual registers (variables) and determines when a value must be placed in a register and when the register can be overwritten with another. When there are more values that need to be held than the amount of available registers, the spilling code is generated. Spilling

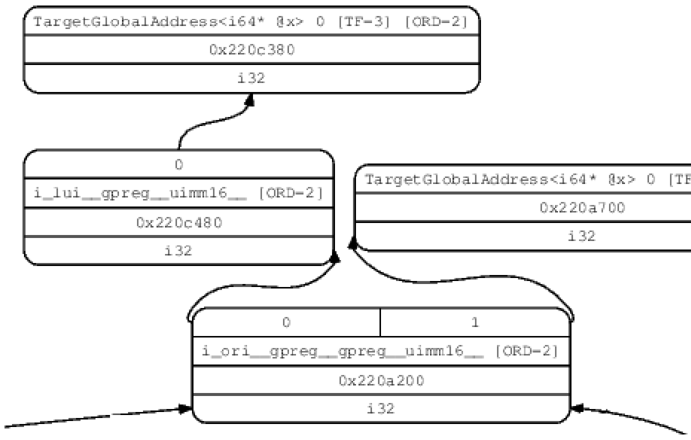

TargetGlobalAddress<i64* @x> 0 [TF=3] [ORD=2]
0x220c380
i32
0
i_lui__gpreg__uimm16__ [ORD=2]
0x220c480
i32
TargetGlobalAddress<i64* @x> 0 [TF=2] [ORD=2]
0x220a700
i32
0
1
i_ori__gpreg__gpreg__uimm16__ [ORD=2]
0x220a200
i32

Figure 2.10: Original SD Node GlobalAddress converterd by ISEL pass into a combination of instructions `lui` and `ori`

means storing a value onto stack and restoring it into a register when the value is needed. LLVM uses linear scan and graph coloring-based algorithms for register allocation [54].

The register allocator uses information about defined register classes and also about the calling convention, both of which were shown in section 2.2.3. It also needs explicit information on how to perform spilling and how to move registers from one class to another; LLVM cannot infer this automatically from the instruction selector patterns.

A part of the code in the LLVM Machine Code intermediate representation used right before register allocation is shown below. To make these examples shorter, we will only show one addition, two instructions, `lui` and `ori`, and a `store` instruction that uses the value that the 3 instructions produce. The instruction names from the preceding examples were shortened for better clarity.

```
// Do the addition.
%vreg4<def> = i_add %vreg1, %vreg3

// Put the higher 16 bits of an address into a register, the TF=3
// specifies that the higher 16 bits of global address 'x' will be
// used.
%vreg11<def> = i_lui <ga:@x>[TF=3]

// Put lower 16 bits of an address into a register, the TF=2 selects
// the lower 16 bits.
%vreg12<def> = i_ori %vreg11, <ga:@x>[TF=2]; gpreg:%vreg12,%vreg11

// And finally store value in %vreg4 onto address in %vreg12 + 4,
// the mem:ST4[@x+4] is an memory operand, it contains information
// for alias analysis about the memory accessed by this
// instruction.
i_store__gpreg__offset %vreg4, %vreg12, 4 ; mem:ST4[@x+4]
```

After register allocation the virtual registers are replaced with the physical registers.

```
%regs_6<def> = i_add %regs_3, %regs_5
%regs_5<def> = i_lui <ga:@x>[TF=3]
%regs_3<def> = i_ori %regs_5<kill>, <ga:@x>[TF=2]
i_store__gpreg__offset %regs_6<kill>, %regs_3, 4; mem:ST4[@x+4]
```

Further, there is another scheduler pass run. Finally, after some further optional optimizations, the assembly code is printed.

Assembly printing

The final pass of the backend is assembly printing. The machine IR is printed into a textual assembly file together with global data definitions and optionally with debugging information. An LLVM backend may also choose to skip the assembler by generating the object code directly, using LLVM MC [55]. The direct object code generation is for reasons of compilation performance, but for testing, compiler development, and low level optimizations it is best to have the assembly code in a readable form.

The assembly printer uses information about instruction syntax as shown in Selection Patterns (see 2.2.3) and prints the assembly code. For our example, these are the instructions printed by the assembly printer.

```
ADD R6, R3, R5
LUI R5, $x>>16 &0xffff
ORI R3, R5, $x &0xffff
STORE R6, R3  +4
```

LLVM - Conclusion

In this section, we have described what intermediate representations are used in LLVM during the compilation and how a target architecture is specified using the C++ code and *tablegen* language. The LLVM implementation is well documented and also the IR tends to be rather readable. LLVM is also written in C++ , which allows much better code structuring and design compared, for example, to the pure C language. The next compiler platform that will be described here is GCC.

2.2.4 GCC

The GNU compiler collection (GCC) is the most widespread retargetable compiler. The very first release was made in the year 1987 [117]. Since then, it has been ported to target a great amount of architectures with more than 40 architectures that are actively supported in the current version 4.9.

In this section, we will overview the GCC compiler and describe the GCC target architecture description. served document The GCC Internals document [94] served as the main source of information for this section.

GCC frontend

The GCC compiler collection contains several different frontends for C, C++, Objective C, Fortran, Java, Ada, Go, Pascal, and several others. All the frontends generate the intermediate language called GENERIC. The purpose of GENERIC is simply to provide a language-independent way of representing an entire function in trees; it is in fact an AST representation. The GENERIC IR serves as interface between the parser and the optimizer.

```
int x;
void add(int a, int b) {
    x = a + b;
}
```

The GENERIC representation for this function can be pretty-printed like C code (while omitting some details):

```
{
  x = a + b;
}
```

Optimizer and GIMPLE IR

The optimizer first lowers the GENERIC IR into GIMPLE IR. GIMPLE is a three-address representation derived from GENERIC by breaking down GENERIC expressions into tuples of no more than three operands (with some exceptions such as function calls). Similar to the LLVM IR, GIMPLE is an SSA-based intermediate representation.

GIMPLE is used for target- and language-independent optimizations (e.g. inlining, constant propagation, tail call elimination, redundancy elimination, etc.). Much like GENERIC, GIMPLE is a language independent, tree based representation. However, it differs from GENERIC in that the GIMPLE grammar is more restrictive: expressions contain no more than three operands (except function calls), it has no control flow structures, and expressions with side-effects are only allowed on the right hand side of assignments.

For our example, the GIMPLE representation in C-style format is as follows:

```
add (int a, int b)
{
  int x.0;

  x.0 = a + b;
  x = x.0;
}
```

And this is the raw format of the same representation:

```
add (int a, int b)
gimple_bind <
  int x.0;

  gimple_assign <plus_expr, x.0, a, b, NULL>
  gimple_assign <var_decl, x, x.0, NULL, NULL>
>
```

Compared to the LLVM IR, you can see one difference here: in LLVM IR the memory accesses are modeled explicitly as memory loads and stores (storing the addition result into the global variable x). In GCC, there is no explicit distiction between memory and register accesses. There is probably a historical reason for this, because GCC started when CISC (Complex Instruction Set Computer) machines were the most widely used machines and RISCs (Reduced Instruction Set Computer) only started to gain in popularity.

Backend and RTL IR

The last part of the GCC compiler work is done on a low-level intermediate representation called Register Transfer Language (RTL). The same name RTL is also used with a different meaning for hardware description languages, but in this chapter, by RTL is meant only the GCC intermediate representation. In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does.

RTL is inspired by Lisp lists. It has both an internal form, made up of structures that point at other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.

All passes in the GCC backend use RTL and those that follow use the so-called machine description and we will review this architecture model used by GCC backends.

A machine description has two parts: files of instruction patterns ('.md' files) and a C header files of macro definitions. The '.md' file for a target machine contains a pattern for each instruction that the target machine supports (or at least each instruction that is worth telling the compiler about). The machine description is described in [94] and [33], but these documents do not contain very good examples, so we will go through the most important parts of the definition here and show it on our running example from the preceding sections.

Expand Pass

The first pass that creates RTL is called *expand*, where based on machine description is each instruction from GIMPLE IR is transformed into either one or multiple RTL operations matching the target instructions.

Information for instruction expansion is specified with the definitions `define_insn` and `define_expand`. If a `define_insn` construction is used, the given template is inserted into the instructions list. If a `define_expand` is used, one of three things happens, based on the condition logic. The condition logic may manually create new instructions for the instructions list. For certain named patterns, it may invoke FAIL to tell the compiler to use an alternate way of performing that task. If it invokes neither DONE nor FAIL, the template given in the pattern is inserted, as if the `define_expand` were a `define_insn`.

For instance, this is a definition of all moves of integer values for the Open RISC architecture [23]. Open RISC is a relatively simple architecture similar for example to MIPS. The name of this define `*movsi_insn` is used when performing *expansion* of moves from GIMPLE to RTL.

```
(define_insn "*movsi_insn"
  // Here is the instruction pattern that says that this is a move
  // (set memory or register with diverse input values)
  [(set (match_operand:SI 0 "nonimmediate_operand" "=r,r,r,r,r,m")
        (match_operand:SI 1 "input_operand"       "I,K,M,r,m,r"))]

  // Constraints on input and output operands
  "(register_operand (operands[0], SImode)
   || (register_operand (operands[1], SImode))
   || (operands[1] == const0_rtx))"
  "@
  // Syntax of particular move versions.
  l.addi  \t%0,r0,%1\t   // move immediate   r <- I
  l.ori   \t%0,r0,%1\t   // move immediate   r <- K
  l.movhi \t%0,hi(%1)\t  // move immediate   r <- M
  l.ori   \t%0,%1,0\t    // move reg to reg  r <- r
  l.lwz   \t%0,%1\t      // SI load          r <- m
  l.sw    \t%0,%1\t      // SI store         m <- r
)
```

The characters I (signed 16-bit immediate), K (unsigned 16-bit immediate), M (higher 16 bits of a 32-bit immediate), r (register), and m (memory) specify the constrains of an input, or output operand, these constrains are defined like this:

```
(define_constraint "I"
  ""
  (and (match_code "const_int")
       (match_test "ival >= -32768 && ival <= 32767")))
```

In our example, we need to store a value to a 32-bit global address. There is no single instruction in Open RISC to move a 32-bit global address to a register, so a *define* specification movsi_insn_big is used.

```
(define_insn_and_split "movsi_insn_big"
  // Instruction pattern
  [(set (match_operand:SI 0 "register_operand" "=r")
    (match_operand:SI 1 "immediate_operand" "i"))]

  // Constraints
  "GET_CODE (operands[1]) != CONST_INT"

  // Syntax - 2 instructions are used for this one operation,
  // in RTL this is represented by one RTL instruction, which is
     then printed
  // as 2 assembly instructions.
  "l.movhi \t%0,hi(%1)\;l.ori   \t%0,%0,lo(%1)"

  // Custom code
{
  if (!or1k_expand_symbol_ref(SImode, operands))
    {
      emit_insn (gen_movsi_high (operands[0], operands[1]));
      emit_insn (gen_movsi_lo_sum (operands[0], operands[0],
         operands[1]));
    }
  DONE;
})
```

The *expand* pass then uses these definitions when *lowering* the GIMPLE IR into RTL, the resulting RTL for for Open RISC follows.

```
// Get an address of a symbol "x" into virtual register reg/f:SI 44
// using the movsi_insn_big pattern.
(insn (set (reg/f:SI 44)
        (symbol_ref:SI ("x")  <var_decl x>))
     (nil))

// Now do the addition with its result in virtual register
// reg:SI 46.
(insn (set (reg:SI 46 )
        (plus:SI (reg/v:SI 42 )
            (reg/v:SI 43)))
     (nil))
```

```
// And finally store reg:SI 46 onto address in reg/f:SI 44
// with the *movsi_insn pattern.
(insn (set (mem/c:SI (reg/f:SI 44) )
        (reg:SI 46))
     (nil))
```

For our running example, this is the RTL representation of the same code for the x86 architecture created by the *expand* pass. You can see that only one RTL operation is needed to store a value to a global address for the x86.

```
// Do the addition operation and store result into
// a virtual register reg:SI 62.
(insn (parallel [
            (set (reg:SI 62)
                (plus:SI (reg/v:SI 60)
                    (reg/v:SI 61 [ b ])))
            (clobber (reg:CC 17 flags))
        ])
     (nil))

// And now we store the previously computed value into memory
// at address of a symbol "x".
(insn (set (mem/c:SI (symbol_ref:DI ("x")
            <var_decl x>))
        (reg:SI 62))
     (expr_list:REG_EQUAL (plus:SI (reg/v:SI 60)
            (reg/v:SI 61))
        (nil)))
```

The x86 version also contains information that the first instruction will clobber (set or invalidate) some *carry* register. Also the x86 has instructions that can contain a whole global address in their binary coding, so there is no need to have separate instructions to get a global address into a register.

Following Machine-Dependent Transformations

Once the instruction list has been generated by *expand*, various optimization passes convert, replace, and rearrange the instructions in the instructions list. This is where the `define_-split` and `define_peephole` patterns come to be used.

Register allocation is also done here. Register classes and register usage are defined with a set of C preprocessor macros. In this way, it is for example defined that Open RISC has 32 general-purpose registers (ARG_POINTER_REGNUM). Also that some of the registers have fixed usage and should not be used by the register allocator.

```
#define OR1K_LAST_ACTUAL_REG        31
#define ARG_POINTER_REGNUM      (OR1K_LAST_ACTUAL_REG + 1)

#define FIXED_REGISTERS { \
  1, 1, 0, 0, 0, 0, 0, 0, \
  0, 1, 1, 0, 0, 0, 0, 0, \
  1, 0, 0, 0, 0, 0, 0, 0, \
  0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1 }
```

One interesting thing in GCC is the absence of an instruction selector working over a DAG or a tree as is usual in algorithms such as BURG [36]. The closest to the instruction selection is the *expand* pass. Optimizations usually done in the instruction selector are in GCC replaced with additional RTL optimizations such as instruction combining and peephole optimizations, which can use more complex instructions instead of the basic ones.

The code is internally represented as RTL until the very end; there the patterns are again matched against the instruction definitions, and the final assembly code is printed:

```
l.movhi r5,hi(x)    # r5 = x << 16
l.ori   r5,r5,lo(x) # r5 = r5 | ( x & 0xFFFF)
l.add   r3,r3,r4    # r3 = r3 + r4 (r3 and r4 are function arguments)
l.sw    0(r5),r3    # store r3 to address from r5
```

The RTL representation tends to be very general so that the target instructions do not appear in it explicitly (except for some special cases, builtins, and inline assembly). This may be an advantage for retargetable machine-dependent optimizations, but writing some specific optimizations that work with particular instructions may not be so straighforward compared to the case where target instructions are in some other compiler machine IR specified explicitly.

GCC - Conclusion

GCC is a very mature compiler collection and its frontends and accompanying libraries are always the ones that are kept up-to date with the latest standards of Open CL [102], Open MP [14], C++11 [52] and other standards that extend the C and C++ languages. For example, the LLVM compiler frontend does not yet officially support Open MP at all. Also, the GCC user base is the largest and development is very active.

On the other hand, due to its historical burden and usage of the C language with a set of sometimes cryptic macro definitions, the code is hard to maintain and modify. Also the RTL IR and machine description format could be better readable. For example, the data dependencies between instructions are well hidden by using only numbers.

Together with LLVM, GCC was one of the compilers considered to be used as a base for the generated compiler; a more detailed comparison follows in section 2.2.8.

2.2.5 SUIF

SUIF is a platform for research into compiler techniques for high-performance machines [118], [63]. The main research topics were scalar data flow optimizations, array data dependence analysis, loop transformations for both locality and parallelism, software pipelining, and instruction scheduling.

The SUIF compiler is structured as a small *kernel* with a toolkit consisting of various analyses and optimizations built using the kernel. The kernel is designed mainly to: 1) make all program information necessary for scalar and parallel optimizations easily available, 2) foster code reuse, and 3) support experimentation and system prototyping.

The kernel performs three major functions: 1) defines intermediate representation of programs, 2) supports a set of IR manipulation primitives, and 3) structures the interface between different compiler phases.

The SUIF compiler source code is distributed with a permissive open source license similar to the BSD license.

SUIF IR

The intermediate representation used in the SUIF compiler is a mixed-level program representation [118]. Besides the conventional low-level operations, the IR includes three high-level constructs: loops, conditional statements, and array access operations. These high-level constructs capture the information useful for parallelization. This approach reduces all the different ways of expressing the same information to a canonical form, thus simplifying the design of the analyses and optimizations. High-level constructs are later lowered into the „low-SUIF“ IR level, where only elementary operations are used.

The SUIF IR also includes detailed symbol and type information, complete enough to translate SUIF IR back into a legal and high-level C code. This can be used to check the transformation correctness, and also to compile the C code, using a different C compiler such as GCC to emit the assembly code for a particular architecture.

This is quite different from LLVM, where a conversion from LLVM IR to C language is also possible using a C code generating backend, but the output code is mostly illegible.

We will show an example of both high-level and low-level SUIF [63].

```
int A[10];
int i;

int fun(int a, int b, int c)
{
    A[i] == a < b ? 10 : 20;
}
```

The high-level SUIF IR printed as C code looks as follows, the conditional expression is still in its original form.

```
int fun(int a, int b, int c)
{
    int suif_tmp0;

    if ( a < b )
        suif_tmp0 = 10;
    else
        suif_tmp0 = 20;

    A[i] == suif_tmp0;
}
```

In the low-level SUIF IR the conditional execution is replaced with jumps and labels, and the array access is lowered. From the 2 assignments to suif_tmp0, you can see that this is not an SSA-based representation.

```
int fun(int a, int b, int c)
{
    int suif_tmp0;

    if ( a < b )
        goto L1;

    suif_tmp0 = 10;
    goto __done2;
L1:
    suif_tmp0 = 20;

__done2:

    *(int *)((char *)A + i * 4) == suif_tmp0;
}
```

Machine SUIF

The SUIF compiler was originally designed in such a way that the user could add their own backend, and no support for backend retargetting was available; later on, a project called Machine SUIF was started, which aims to fill this gap [99], [8].

It complements SUIF with low-level but still partially architecture-independent IR, which shows one-to-one correspondence to assembly instructions. The goal is to keep most of the source code machine-independent and encapsulate the architecture-specific details in a target library [63].

Machine SUIF uses SUIFvm representation/architecture as an intermediate target in translation from the SUIF 2 form into instructions for a real target machine [37].

Operations in SUIFvm are quite similar to the basic LLVM SD Nodes operations, with some notable differences:

- NOP is an operation that has no effect,
- MEMCPY allows copying a value from one memory position to another, such an instruction is not usual in load/store architectures,
- MOV moves one value from one virtual register to another, in LLVM SD Nodes, moves are not necessary, because before instruction selection all registers are virtual and the code is in the SSA form,
- MBR is a multi-way branch instruction that provides a behavior similar to the switch C language construct; such instructions do not exist in standard architectures, and
- there is no distinction between floating-point and integer operations, meaning that the instruction operands determine the type of operation.

Retargeting is done by implementing a function that replaces each of these 46 SUIFvm operations with target instructions. This approach is quite similar to the GCC expand pass. An example of a function for translating the MOV instruction for the x86 architecture is shown here:

```
void CodeGenX86::translate_mov(Instr *mi) {
    if (is_floating_point(get_type(get_dst(mi)))) {
        // Opcode is null because copy is handled by
        // src/dst processing.
        translate_fp(opcode_null, mi);
        delete mi;
    } else {
        set_opcode(mi, x86::MOV);
        emit(mi);
    }
}
```

Automatically generating the Machine SUIF backend may seem to be fairly very simple at first glance. In fact, only an equivalent to each of the 46 operations needs to be found. However, each operation can have different operand types, and a complex code for operation lowering must be added by hand.

Not using an instruction selection algorithm limits the number of instructions the backend can effectively use. Adding a very smart peephole optimizer that cleans up the code after the transformation from SUIFvm into machine instructions could improve the performance.

Due to this inefficient code generation, the resulting code will be slow, e.g. as reported in [32], where the authors reported a twofold slowdown for the ARM v5 architecture.

Before LLVM gained its popularity, SUIF was the most interesting research platform for statically compiled languages, but currently it is not maintained anymore.

Machine SUIF compiler source code is distributed with a permissive open source license similar to the BSD license.

2.2.6 CoSy and Backend Generation from LISA ADL

The CoSy compiler development system is a retargetable compiler development system from ACE, Associated Compiler Experts [1], which has been deployed in a broad spectrum of targets, ranging from 8-bit microcontrollers to CISC, RISC, DSP and 256-bit VLIW processor architectures. The CoSy compiler development system contains a frontend, architecture-independent optimizer, and retargetable backend. In this section, we will focus on the backend, and also on its generation from an architecture description language LISA [35]. Used as the main source of information was the book [36], especially its chapter 6.

CoSy Instruction Selector

Instruction selector in the CoSy backend uses a dynamic programming tree-matching algorithm, based on algorithm described in [2].

The instruction definition that is used to generate the instruction selector looks like this:

```
// Instruction selection rule, the mirPlus is the operator,
// a, b, and c are operands.
RULE o:mirPlus (a:reg_nt, b:reg_nt) -> c:reg_nt
// Constraint for applying this rule.
CONDITION { IS_INT(o) }
// Cost of instruction.
COST 1;
// Code for assembly printer.
EMIT {
  Print("add %s = %s, %s", REGNAME(c), REGNAME(a), REGNAME(b));
}
```

The instruction selector generator then collects all these rule definitions and constructs an automaton that then parses the input selection DAG and, using dynamic programming, matches the instructions. The internal CoSy representations and function have not been published, so we cannot show many more details here, but what is interesting as regards the theme of this thesis is the automatic CoSy backend generation.

Extensions in LISA for Compiler Generation

LISA is a mixed architecture description language (see 2.1.2) and the accompanying tools allow automatic CoSy backend generation.

As in CodAL, instruction behavior is defined using the C language. The authors of the compiler generator state in [36] that the *semantic gap* between the description of instructions in the C language and the form needed by the instruction selector is too wide and that the C language description is too „informal“ to be used. Because of this reason, they introduced a second description of instruction behavior into the LISA language, with a whole new definition language. For example, an addition instruction that uses 1 output register, 1 input register, and either 1 input register or 1 immediate operand is shown here:

```
OPERATION ADD {
  // Declare local instances of other operations.
  DECLARE {
    GROUP src1, dst = { reg };
    // Instance src2 can be either a register, or an immediate.
    GROUP src2 = { reg || imm };
  }
  // Assembly syntax.
  SYNTAX { "ADD" dst "=" src1 "," src2 }
  // Binary coding.
  CODING { 0b0000 src1 src2 dst }
  // Instruction behavior for simulator in C language.
  BEHAVIOR {
    dst = src1 + src2;
    carry = (uint32)(src1 + src2) < (uint32)src1;
  }
  // Semantics for compiler generation using
  // micro-operations.
  SEMANTICS {
    _ADD|_C|(src1, src2)<0, 32> -> dst;
  }
}
```

A micro-operation in the SEMANTICS section is a tuple *(o, S, U, v, w)* consinsting of the micro-operator o (_ADD), set of side effects S (carry flag - _C), set of operands U (src1, src2, and dst), and bitwidth w (<0,32>).

As another example, we will show the semantics of a multiply-and-accumulate instruction:

```
_ADD(_MULUU(src1, src2)<0, 32>, dst) -> dst;
```

And this is the semantics of an instruction that swaps 16 bits in a 32-bit register:

```
src<0,16> -> src<16,16>;
src<16,16> -> src<0,16>;
```

To express loads and stores, the `_INDIR(address)<bits>` micro-operator is used that can be both read and written. Jumps are then expressed by writing to a special register `_PC` that represents the program counter.

CoSy Instruction Selector Rules Generation

The instruction selector generator keeps a so-called *basic library* containing the basic instruction selection rules needed for a complete coverage of C operations. For each basic rule in the library, a list of target-specific tree patterns is generated. For example, for the `mirPlus` and `mirDiff` operations from the CoSy IR matched with the corresponding LISA micro-operation, either a register or an immediate is substituted for each pattern operand, as shown in Figure 2.11.

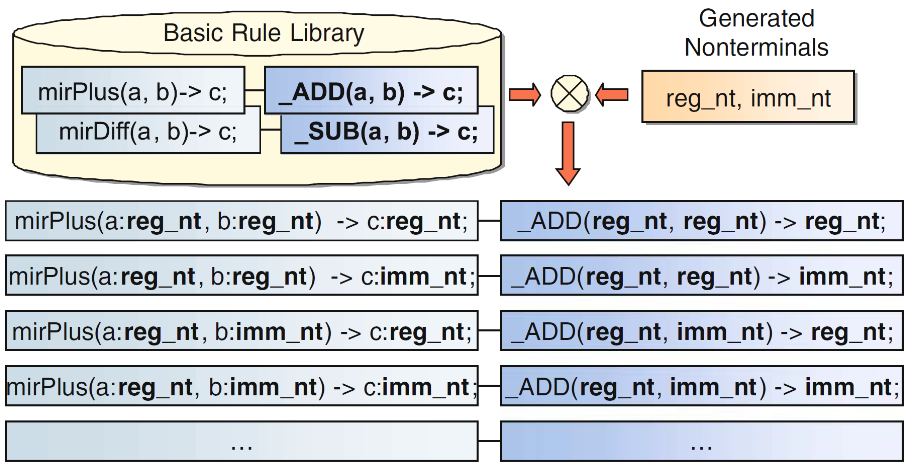

Basic Rule Library
mirPlus(a, b)-> c;
_ADD(a, b) -> c;
mirDiff(a, b)-> c;
_SUB(a, b) -> c;
Generated Nonterminals
reg_nt, imm_nt
mirPlus(a:reg_nt, b:reg_nt) -> c:reg_nt;
_ADD(reg_nt, reg_nt) -> reg_nt;
mirPlus(a:reg_nt, b:reg_nt) -> c:imm_nt;
_ADD(reg_nt, reg_nt) -> imm_nt;
mirPlus(a:reg_nt, b:imm_nt) -> c:reg_nt;
_ADD(reg_nt, imm_nt) -> reg_nt;
mirPlus(a:reg_nt, b:imm_nt) -> c:imm_nt;
_ADD(reg_nt, imm_nt) -> imm_nt;
...
...

Figure 2.11: Tree pattern generation in LISA-based compiler generator [36]

Once all the target-specific rules have been generated, and useless rules have been filtered out (such as those which write to an immediate operand), the next task is to find suitable instructions in the LISA model that match the semantic statements of the generated tree patterns. There are now 3 different mapping types that can be produced:

- *one-to-one* mapping is used when the generated rule can be matched exactly by on target instruction,
- *one-to-many* mapping implements one rule with multiple instructions, and
- *many-to-one* mapping is rather an optimization and is used when an instruction can execute multiple elementary operations such as MAC (multiply and accumulate).

The *one-to-many* mapping generation uses a library of mathematically equivalent solutions to perform a selected operation. This library can be extended by the user. Once the instruction selector rules and other auxiliary operations such as register moves, conditional jumps, store/reload the stack pointer on/from the stack, increase/decrease the stack pointer by an immediate value, and indirect jump (return) have been generated, the CoSy compiler can be built and used.

The approach taken in LISA is to use additional information in the ADL model to express instruction behavior for the compiler, which makes the compiler generation fairly automatic. However, having 2 descriptions of instruction behavior even in an instruction-accurate model can introduce inconsistencies. The author could not find how this inconsistency problem can be approached; probably the only way to check the equivalence of these two descriptions is through simulation and testing.

Current Status of LISA-based Compiler Generator

Due to the competition mainly from LLVM, the ACE company does not focus on CoSy compiler development so much (the last release was in the spring of 2013). They are currently switching to LLVM and start to offer LLVM maintenance and support for their customers. ACE also offer a large SuperTest compiler test and validation suite [6] that is the largest offered C and C++ compiler testsuite on the market.

The LISA language was originally developed at the RWTH Aachen University. This research project then transformed into a spin-off LISATek. LisaTek was later acquired by CoWare, a virtual platform design tool developer. CoWare was later acquired by Synopsys.

The current solution based on LISA employed in the Synopsys Processor Designer [107] uses LLVM as a compiler platform and requires the user to make many manual modifications of the LLVM source code. These modifications are well documented, but still require a developer who knows the LLVM compiler platform.

2.2.7 Other Retargetable Compilers

Trimaran

The Trimaran system, developed primarily at HP Research Labs, is an extensible compiler framework for research into code optimization techniques with a philosophy similar to SUIF [62] but with focus on optimizations for instruction-level paralellism, and therefore it supports only a specific class of parameterizable VLIW processors [60].

Open64

Open64 [84] was a compiler originally targeting the Intel Itanium VLIW architecture. Open64 also derives from the work done by Intel Corp, in conjunction with the Chinese Academy of Sciences. They created the Open Research Compiler (ORC), and all their changes were later included in the Open64 compiler sources. Although it was not originally started as a retargetable compiler, some retargetability is available, for example as described in [64]. The Open64 compiler is published under the GPL v2 license, contains the most advanced VLIW optimizations and was often used as a research platform for new optimizations. The last official release of Open64 was in the year 2011.

Chess/Checkers

The Chess/Checkers Compiler framework is used by the company Target [106] for C compiler generation from ADL nMl [79]. The Target Company has recently (spring 2014) been acquired by Synopsys. There are several publications on C compiler generation from ADL nMl, but none of them describes the process used to generate a compiler in the Target's IP Designer, and Target does not publish such information. For example, one article describes the generation of a backend for the LLC compiler [80]. LLC is an academic project to make the compiler as simple as possible, and definitely not usable for efficient code generation. Another article [87] describes the generation of the GCC compiler from the similar Sim-nMl language.

2.2.8 Retargetable Compilers - Conclusion

We described several retargetable compilers. At the beginning of the design and implementation of a C compiler generator from CodAL, we needed to choose a base compiler platform. We were deciding between LLVM and GCC, because SUIF and Trimaran are not actively maintained. The CoSy compiler is a proprietary code with an expensive developer license. The Chess/Checkers compiler is also proprietary and a license for the source code is not offered.

LLVM was chosen mainly due to its newer and cleaner implementation, better documentation, and better readable internal representations.

An analysis and a comparison were made in [110]. In favor of LLVM was the simplicity of its maintenance, and the slightly more modern algorithms used in the backend. In favor of GCC was its performance, stability, and availability for more targets.

A very important consideration was also the compiler source code license. LLVM is provided with a less restrictive LLVM Release License, which does not require publishing the modified source codes included the libraries unlike the GCC's General Public License (GPL). At Lissom and Codasip, we made many extensions to the LLVM base source code over time. Using the GPL license would mean that the added extensions would not provide a competitive advantage, because with GPL, the source code would need to be made publicly available.

2.3 Reconfigurable Processors

Another part of the envisioned ASIP optimization tool together with the compiler generator is a processor template. This section describes two user-reconfigurable processors, Tensilica Xtensa and Synopsys ARC. Both processors can be extended with custom instruction set extensions and differ in the degree of configurability and in the support of the programming tools for the new extensions.

2.3.1 Xtensa

Xtensa [31] is an extensible processor template developed by Tensilica [96], where the user can specify new instructions using a TIE language (Tensilica Instruction Extension). Xtensa is now provided by Cadence. Some features of the processor can be enabled or disabled such as floating point support, interrupts support, memory ordering, register file size, etc. It is also possible to enable VLIW (for example [27]) execution units, this option for Xtensa is called VLIX. Figure 2.12 shows the high-level structure of the Xtensa processor core.

TIE lets the designer specify the mnemonic, the encoding, and the semantics of single-cycle instructions. The new instructions are then placed into the *Designer-defined instruction execution unit.*

An example of a `BYTESWAP` instruction that swaps bits and accumulates 10-bit values follows:

```
// Define a new opcode for byteswap.
opcode BYTESWAP op2=4'b0000 CUST0

// Declare state SWAP and ACCUM.
state SWAP 1
state ACCUM 40

// Map ACCUM and SWAP to user register file entries.
user_register 0 ACCUM[31:0]
user_register 1 {SWAP, ACCUM[39:32]}

// Define a new instruction class.
iclass bs {BYTESWAP}
```

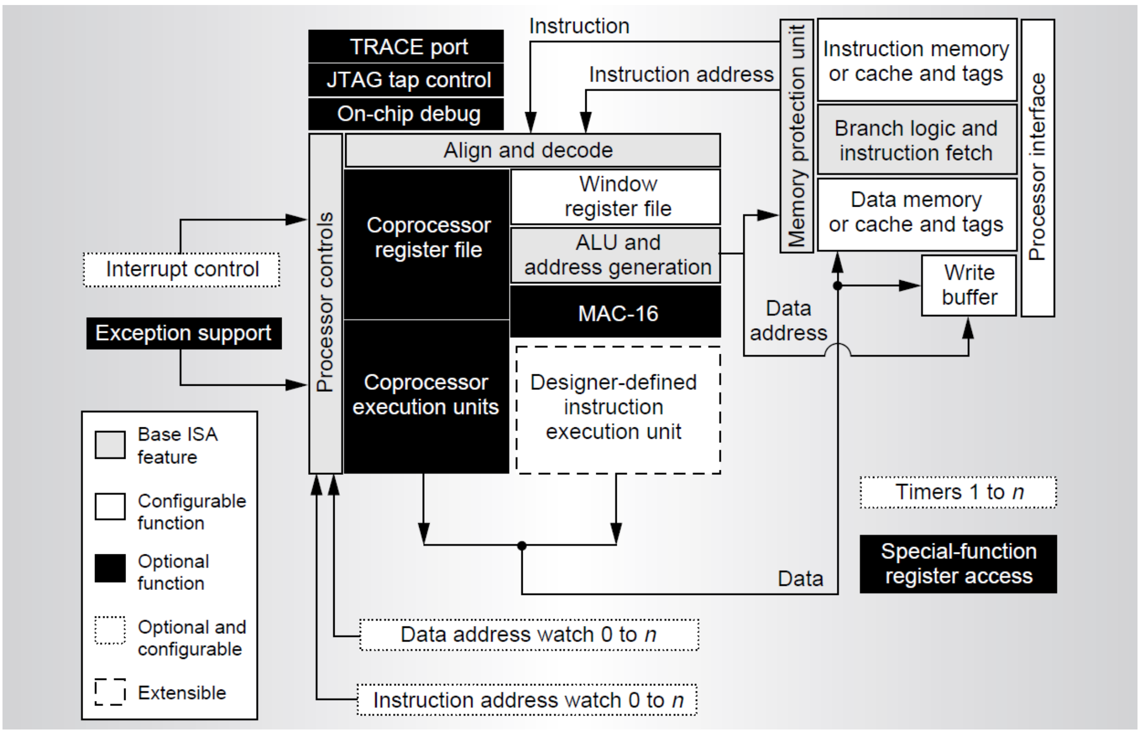

Instruction
TRACE port
JTAG tap control
On-chip debug
Instruction address
Memory protection unit
Instruction memory or cache and tags
Branch logic and instruction fetch
Data memory or cache and tags
Write buffer
Processor interface
Align and decode
Window register file
ALU and address generation
MAC-16
Coprocessor register file
Coprocessor execution units
Designer-defined instruction execution unit
Processor controls
Interrupt control
Exception support
Data address
Data
Timers 1 to n
Special-function register access
Data address watch 0 to n
Instruction address watch 0 to n
Base ISA feature
Configurable function
Optional function
Optional and configurable
Extensible

Figure 2.12: Block diagram of the Xtensa processor [31]

```
  { out arr , in ars }
  { in SWAP , inout ACCUM}

// Semantic definition of byteswap.
semantic bs {BYTESWAP} {
  wire [31:0] ars_swapped =
  {ars [7:0] , ars[15:8] , ars [23:16] , ars [31:24]};
  assign arr = SWAP ? ars_swapped : ars ;
  assign ACCUM = {ACCUM[39:30] + arr[31:24] ,
  ACCUM[29:20] + arr[23:16] ,
  ACCUM[19:10] + arr[15:8] ,
  ACCUM[ 9: 0] + arr[7:0]};
}
```

New state registers are defined with the `state` declaration. The TIE compiler generates additional instructions that allow moving values from and to these new registers. The `iclass` is the definition of an instruction class containing one `BYTESWAP` instruction. Instructions in this class use `arr` as the output, `ars` as the input, the `SWAP` user state register as the input, and the `ACCUM` user state register as both the input and the output. The `arr` and `ars` inputs are values passed through a side-bus that connects the base processor core and the extension unit.

The behavior of the new instruction (`BYTESWAP`) is described in the semantic block. The instruction conditionally reverses the byte order of the input operand and accumulates the values of the different byte lanes [31].

The TIE compiler also automatically adds the new instructions to the RTL, and to the software tools including the compiler and assembler. When the semantic description is too complex to be used by the compiler, another type of description can be used for the compiler generator.

The compiler also translates the instruction semantics into the appropriate hardware description language and produces the appropriate pipeline control signals. It automatically generates the bypass control, interlock detection, and immediate generation logic required by the instructions.

Tensilica Xtensa is a processor successfully used in wired and wireless networking, printers, home entertainment, and other application domains. Compared to other ADL-based approaches, the base architecture is fixed (only with some optional features). Extensions can be specified by the TIE language and then the whole toolchain and RTL can be regenerated.

2.3.2 ARC

ARC is another configurable processor core, originally developed by ARC International, and is provided by Synopsys as DesignWare ARC processors.

ARC cores are provided in multiple versions with 3- (ARC EM), 5- (ARC 600), 7- (ARC700), and 10-stage (ARC HS) pipelines. The ARC HS pipeline is shown in Figure 2.13. Target applications differ for each of the versions, the fastest ARC HS and ARC 700 are targeted at digital TVs, set-top boxes, baseband control, and home networking. The ARC 600 and ARC EM versions are, for example, used in solid state device controllers, wireless LANs, and sensors. It is also targeted at wearable devices.

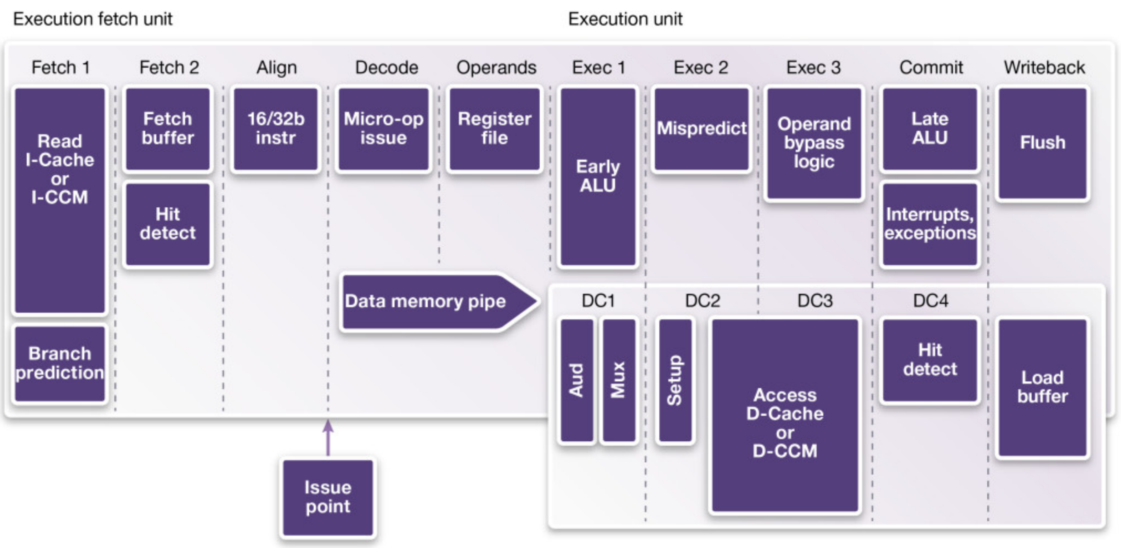


Figure 2.13: Scheme of the ARC HS processor core pipeline [109]

The ARC core can be customized with DesignWare ARChitect IP Configurator as is shown in Figure 2.14.

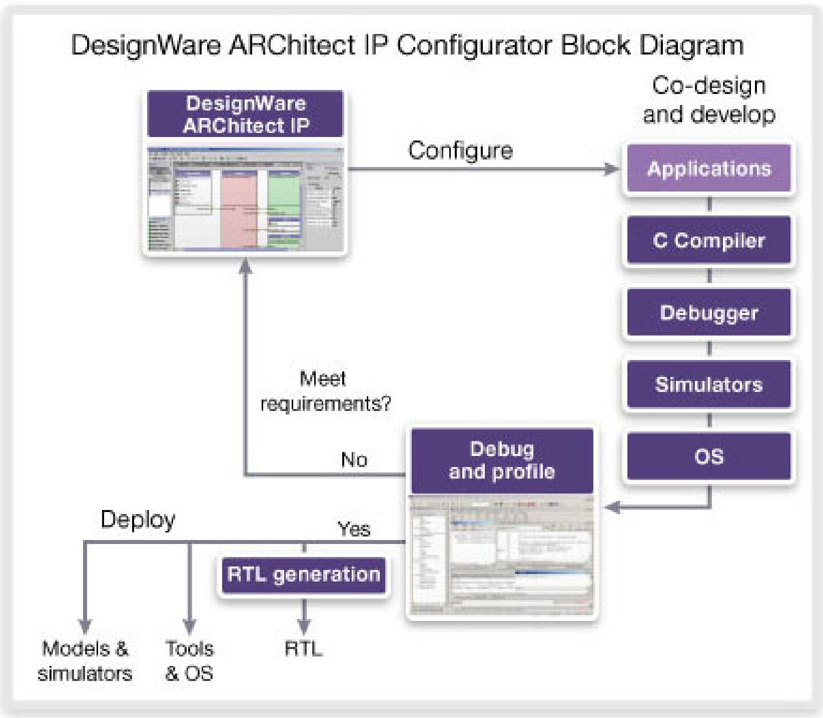


Figure 2.14: Design methodology with DesignWare ARChitect IP Configurator [105]

Drag-and-drop menus are used to select configuration, among configuration options are MMU, single, or double precision floating point, configurable memories and caches, DSP instructions, real-time tracing support, and security algorithms acceleration extensions [104]. The user can also create custom instruction extensions. Custom instructions are added through an interface om the Exec 1 stage. In Figure 2.14 is shown a design methodology usable with tools for ARC processors.

The ARC core can be customized with DesignWare ARChitect IP Configurator, as shown in Figure 2.14. Drag-and-drop menus are used to select configuration; among the configuration options are MMU, single- or double-precision floating point, configurable memories and caches, DSP instructions, real-time tracing support, and security algorithm acceleration extensions [104]. The user can also create custom instruction extensions. Custom instructions are added through an interface of the Exec 1 stage.

Custom instruction support in programming and simulation is quite limited, because only the drag-and drop configurations control the C compiler, simulator, and RTL. When a new instruction is added, it must be described in at least three different representations. The first one is (optional) modification of the GCC source code. The second is a simulation version either in C/C++ or in System C. Then the last needed representation is hardware description in VHDL or Verilog. It is up to the user to make sure that each of these representations expresses the same instruction.

Even if support for custom instructions is not much automated, the ARC processor is used by more than 170 customers of Synopsys, with more than one billion ARC-based chips shipped annually [103].

2.4 Optimizing Processors with Instruction Set Extensions

In the previous sections, we first explored the problematics of retargetable compilation where a higher language compiler can be adapted for different architectures. Support for new instruction set extensions (ISEs) can be easily added to retargetable compilers. Then we described two commercially used processors that can be extended with new ISEs.

Now, when we want to optimize an extensible processor, we must identify what new instructions should be included in the design. There are manual and automatic approaches to this problem and both will be reviewed in this section.

2.4.1 Manual ISE identification

Manual ISE identification takes advantage of human ability to spot interesting parts of the application that can be implemented in hardware. An example of manual ISE identification usable with ADL-based tools consists of the following steps:

1. Profile the application compiled with maximal optimization to find the parts of code that take the most of the execution time (application kernels). The profiling granularity of functions may not be sufficient due to inlining and presence of large functions. Therefore approximate information about cycles spent in each line of the source code is a much better guidance.

2. Optimize the code manually to really see whether the later applied ISE has the desired effect on the target application. It is necessary to optimize the code first in order to see actual improvements.

3. The most time-consuming parts of the code are analyzed. The goal is to find code sections with limited inputs and outputs and model them into a new ISE. Only limited memory accesses can be present in the new function in order to be able to generate efficient hardware. Some memory loads represent a search in a constant lookup table. If the calculation can be done efficiently in hardware, the lookup table can be transformed into a computation. For other cases, memory access can be transformed into a hardware lookup table. Before the speed-up provided by this ISE is known, the ISE selection is constrained neither by maximal area nor by the maximum of register operands.

4. The ISE found is described in an architecture description language. If the amount of input-output registers is higher than the ports provided by the register file, then the remaining register accesses are modeled as accesses to fixed registers. Addresses of fixed registers are specified in instruction opcode. For example, on an FPGA with register field implemented from LUTs, an arbitrary number of fixed registers can be read and written simultaneously even if only 1 write and 2 read ports are available. Once the ISE has been described, the toolchain is regenerated, so the assembler, simulator, and the C compiler know about the new instruction.

5. The original block of code, which is now implemented in the new ISE, is replaced with an inline assembly with input register operands and constraints on the fixed registers described. The impact of the new ISE on the application performance is measured and the designer may decide whether the cost of the new instruction justifies the performance improvement.

6. If the resulting system does not satisfy the requirements, the application is profiled again and steps 3 - 6 are repeated until a solution that satisfies requirements is reached.

The main advantage of manual ISE identification consists in finding large parts of code that cannot be found with automatic approaches. This mainly concerns larger sets of basic blocks that the programmer identifies as implementable using combinational logic. Also some special optimizations with lookup tables, local memories and special registers can be applied. The disadvantage is that most of the manually identified ISEs are too complex to be matched with classical instruction selection algorithms and can be used only for a small set of applications. Other algorithms for complex instructions or accelerator usage in a compiler exist that can be used practically even if the theoretical complexity of acyclic graph covering is exponential [18]. The manual ISE identification approach may be also time-consuming for large applications.

2.4.2 Automatic ISE Identification

Automatic ISE identification algorithms use as the input a program in some control-dataflow graph representation (usually a compiler's intermediate representation), profile, and microarchitectural constraints such as maximal area, or available register file ports. In a thorough survey of the ISE-related problematics [28] the authors of this survey distinguish two basic approaches based on the granularity at which the code is considered. Fine-grained approach works at the operation level and implements small clusters of operations. Coarse-grained approach operates on the loop or function level and can provide higher speed-up at the expense of specialized ISEs that can be used only for a specific application.

Since the total space that can be searched is exponential [28], techniques for either reducing the complexity or pruning the search space must be used. Generally, the ISE identification algorithms search for convex cuts (subgraphs that have no inputs that come initially as outputs from the selected cut). Several approaches to automatic ISE identification are described here.

In [19], the authors analyze a dataflow graph of the target instruction before scheduling and register allocation. For hardware cost and timing, a hardware library is used. They first collect a set of subgraphs with a guide fiction that mainly considers position on the critical sequential path, latency, area, and input/output constraints.

The collected set of subgraphs is then passed to a candidate combination stage. This stage groups subgraphs that can be executed on the same piece of hardware. Grouping the subgraphs creates a set of candidate custom function units and allows calculating an estimate of performance gain by using the profile weights of all the set members.

The functional units found are then represented as instructions in the hardware and are used in the instruction selection pass of a compiler backend. The authors do not consider memory operations. In [88], the authors propose two kinds of solution to solve the ISE identification problem: exact and approximate methods. The approximate methods are used when the size of basic blocks prevents the exact method to terminate. The exact method works as follows: 1) find the optimal single cut in a single basic block; 2) find the optimal set of nonoverlapping cuts in a single basic block; and 3) find an optimal set of nonoverlapping cuts in several basic blocks.

The approximate methods use either pseudooptimal selection, where as a guide for the selection are cuts with the highest speedup used. The first method makes an iterative selection where the single-cut identification algorithm is iteratively applied to the same basic block. The previously identified cuts are excluded from the next iteration. The partition-based selection method prefers partitions with the smallest number of inputs and outputs. The genetic algorithm method uses a subset of nodes representing one solution. One solution is encoded as a binary string and genetic operations are applied to a population of solutions.

In [11], the authors build on methods presented in [88] and consider a local scratchpad memory for ISEs. Their ISE identification works as follows: 1) find vectors and scalars accessed in critical basic blocks, for example by using static memory disambiguation techniques; 2) search for the most profitable code positions for inserting memory transfers between the application-specific functional units; 3) run an ISE identification that is based on [88]. An important part of the scratchpad memory approach is scheduling DMA data transfers and ensuring memory consistency.

In [82] the authors examine conditional execution in hardware to be able to map a subgraph of a control-dataflow graph to hardware. Once such subgraph is identified, they perform temporal partitioning.

And as the last example of ISE identification approaches, in [113] the ISE input/output serialization is considered. This allows overcoming microarchitectural constraints by providing multiple input and output operands. Their approach to ISE identification consists of the following steps: 1) during clustering the nodes are grouped into equivalence classes based on a speed-up model; 2) each equivalence class is then compressed into a single node; 3) optionally, the nodes and edges that do not affect the optimal ISE are removed; 4) each maximal clique in the cluster graph is exhaustively enumerated (a clique is a subset of graph nodes such that every two nodes in the subset are connected by an edge; the maximal clique is a clique that cannot be extended by including one more adjacent node); 5) optionally, the properties of the base processor and the speed-up model are used to prune the cliques found; 6) the input/output accesses of ISEs corresponding to the remaining cliques are serialized and the ISE offering maximal speed-up is selected; the final step 7) consists in reducing the size of the chosen ISE, i.e. finding a smaller ISE that offers the same speed-up.

This process is repeated in order to select multiple ISEs from the same DAG. Each ISE that is identified is replaced with a single node, which is marked as forbidden, and this prevents overlapping ISEs.

Graphs 2.15, and 2.16 show results collected from papers on automatic ISE optimzation. Speedups from articles are collected in the following graphs as A2-A14. A2 represents data from article [61], A3 [19], A4 [17], A5 [88], A6 [72], A7 [70], A8 [73], A9 [116], A10 [10], A11 [29], A12 [11], A13 [113], and A14 [25]. Results for other benchmarks are also listed later in appendix D, where they are compared with manual optimization results. The speedup that can be obtained with the automatic ISE identification methods is usually around 2x. Only for certain applications, it can go up up to 6x.

The main problems in automatic ISE identification arise from computational complexity, where hardware/software partitioning that is equivalent to instruction-set customization is proven NP-hard in the general case. Optimal solutions have been proposed by many authors and a large amount of efficient heuristics was found. However, as stated in [28], quality results are produced through a balance of human intervention and automatic methods.

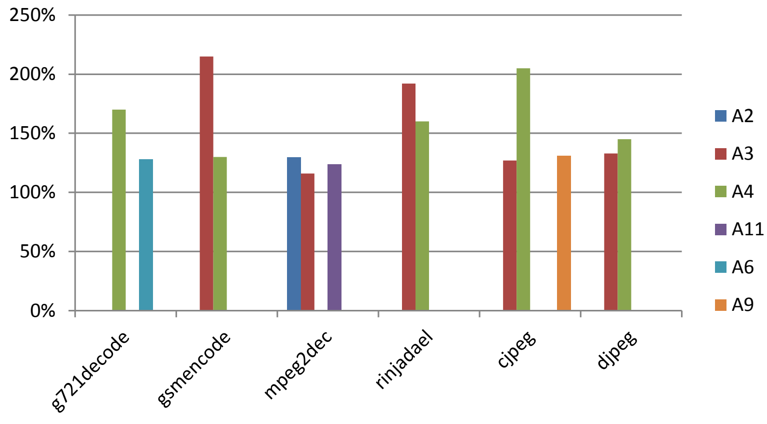


Figure 2.15: Speedups for automatic ISE identification methods (1)

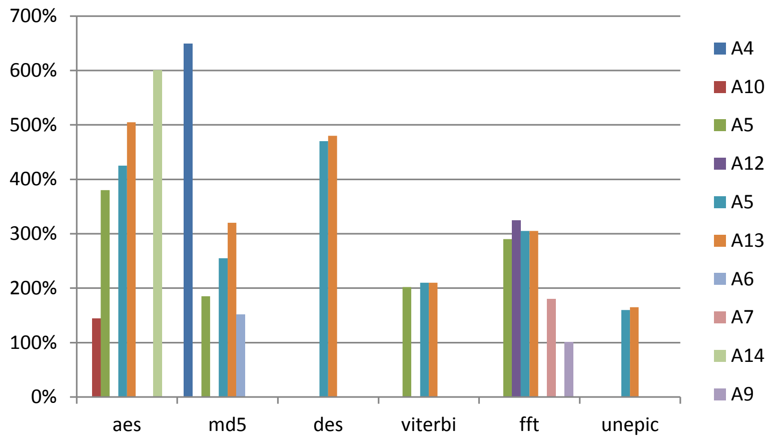


Figure 2.16: Speedups for automatic ISE identification methods (2)

2.5 State of the Art - Conclusion

The planned goal of this thesis, as described in the introduction, is to create a C compiler generator, an extensible ASIP processor template, and methods how to optimize the processor template using the compiler and by adding new instructions. This, together with other tools, will form a complete ASIP design and optimization environment where the user starts with a processor template and optimizes it for their needs. In this chapter were described areas related to the problematics of ASIP design. Also, two examples of reconfigurable processor were described.

To sum up, architecture description languages (ADLs) provide means to describe a processor in an abstract way. From the ADL definition, programming tools can then be generated, including C compiler, assembler, and simulator with debugger. Also, an RTL definition can automatically be obtained from an ADL model.

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

The only commercially used compiler generators based on ADLs use the nMl and LISA languages. Compiler generation from nMl has not been published. In the LISA compiler generator, instructions for the compiler generator are described in 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 then mean that the ADL is kept simpler, and that the inconsistency problem is avoided.

Applications that require higher performance are run usually on 32-bit processors, also 16-bit processors can sometimes be useful. This means that it is unimportant to focus on 8-bit architectures with their often obscure instruction sets and unusual way of addressing, etc. The target architectures for the generated compiler will be 32- and 16-bit RISC architectures.

The reconfigurable processor cores that were described 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. For example, it may be necessary to add reading ports to the register file or to modify the memory interfaces. This is 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 in this way. We do not want the unaccelerated part of the application to take too long, because this would limit the achievable speed-up (by Amdahl's law), so the base instruction set must be very efficient. It is also important that the new processor should be easily extensible. The microarchitecture must be rather simple, so that 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 allow only a 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 their own extensions, and automatize their efforts as much as possible.

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

Chapter 3

Solution

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

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

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

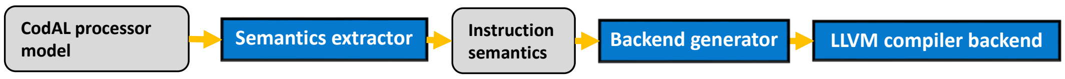


Figure 3.1: Overall structure of the C compiler generator

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 move them automatically into the template of an extensible processor core.

3.1 Instruction Semantics Extraction

For compiler backend generation, the compiler generator needs as the input some kind of analyzable model of instruction set description. The model developed is called instruction semantics model. The *instruction semantics* is primarily used for compiler backend generation, but it has also other uses, as described in Appendix E.

First, we specify some requirements for the instruction semantics model with respect to compiler generation. These requirements follow the LLVM backend target architecture model described in section 2.2.3. The compiler generator must be able to identify several important instructions, the most significant being:

- register moves,
- memory accesses, also with accesses to a stack,

- move of a pointer-wide (e.g. 32-bit) immediate value into a register 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 form to generate instruction selection patterns. 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. Details about the architecture information needed by LLVM were presented in section 2.2.3. In this section, it is described how such a model that satisfies the stated requirements can be obtained from a CodAL model.

3.1.1 Instruction Enumeration and Unoptimized Code Generation

For the purposes of compiler generation the author has designed and implemented a tool called *Semantics extractor*. Its purpose is to analyze the CodAL model, construct a list of all instructions and simplify their behavior so that it can be analyzed for C compiler generation. The resulting format was originally based on LLVM IR [68] and later it was changed to better match the operations used in instruction selection in a LLVM backend (so-called SD nodes). The SD nodes are very similar to the LLVM IR operations.

Instruction Description in CodAL Language

The instruction set description in the CodAL language is based on context-free grammars. For example, the assembler generator uses this fact and a mechanism based on translational context-free grammars is used to transform the instruction assembly form into its binary form.

For a better understanding, an example is used in this chapter. This example shows two instructions, ADD and SUB, with 3 register operands.

The following piece of CodAL code describes the operation codes.

```
element opc_add
{
    assembler {"ADD"};
    binary {0b100000};
    return {0x20;};
}

element opc_sub
{
    assembler {"SUB"};
    binary {0b100010};
    return {0x22;};
}
```

In the case of CodAL elements, the element name is used as a left-hand side nonterminal and the assembler section contents form the right-hand side of the rule. For the elements `opc_add` and `opc_sub` the context-free grammar rules *opc_ add -> "ADD"* and *opc_ add -> "SUB"* were created.

The `set` construction is used to create a set that, when instantiated in an `event`, represents any of the contained elements or other sets.

```
set opc = opc_add, opc_sub;
```

Since any of the nonterminals from the set can be used in place of the main set nonterminal, the behavior of the set construct is described by the rules *opc -> opc_ add* and *opc -> opc_ sub*.

Instruction representation for compiler generation must have its register operands explicitly described. To specify that a set or element is a register operand, a new keyword `represents` was introduced in CodAL. The keyword `represents` is followed by a physical register field name.

```
element gpreg represents regs
{
  assembler  { "R" regnum=unsigned };
  binary     { regnum=0b[5] };
  return { regnum; };
}
```

The element `gpreg` describes a register class called the same name `gpreg`. The contents are inferred from all allowed binary encodings, and the assembly syntax of the registers is *R0, R1, ..., R31*. For the purpose of semantic extraction this register class is represented by a rule *gpreg -> "GPREG"*, where *"GPREG"* is a special terminal representing this register class.

Finally, we can put the already defined elements and sets together and define the whole instruction.

```
element instr {
  use gpreg as rd, rs, rt;
  use opc;

  assembler { opc rd "," rs "," rt };
  binary    { opc rs rt rd 0b000000  };

  semantics {
    switch (opc) {
      case 0x20: regs[rd] = regs[rs] + regs[rt];
        break;
      case 0x22: regs[rd] = regs[rs] - regs[rt];
        break;
  }};
}
```

For a context-free grammar rule, the instances *rd*, *rs*, and *rt* are replaced by the element *gpreg*, and the rule *instr -> opc gpreg "," rs "," rt* is created.

Generating Instructions with Unoptimized Code

Now when we have a context-free grammar describing the syntax of instructions with register operands specified, a set of all instructions can be obtained. If the grammar G contains all context-free rules created from the CodAL description, then the language generated by this grammar $L(G)$ is a set of all instructions described in the CodAL description. Since cycles are not allowed in the CodAL language, the language $L(G)$ is finite.

To obtain the *L(G)*, we create a DAG from the context-free grammar rules. For the purpose of instruction enumeration, this graph can be seen as a finite automaton such as the one in Figure 3.2.

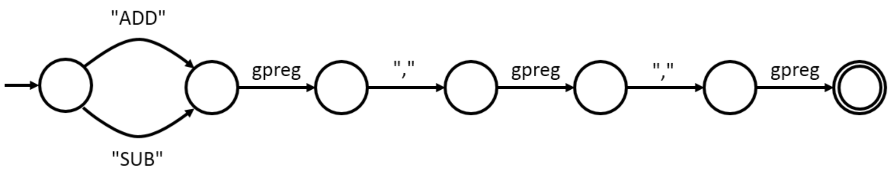


Figure 3.2: Example automaton used by Semantics extractor to enumerate all instructions

Each path from the starting state to one of the final states now represents one word from *L(G)*. Here we get two instructions: `''ADD'' gpreg '','' gpreg '','' gpreg` and `''SUB'' gpreg '','' gpreg '','' gpreg`.

The DAG has semantic information on nodes that is used to generate the instruction semantics code. The following code has been generated for our example of instruction ADD.

```
const int opc = 0x20; {
const int rd = codasip_regopindex(0); {
const int rs = codasip_regopindex(1); {
const int rt = codasip_regopindex(2); {
{
 switch (opc) {
 case 0x20: regs[rd] = regs[rs] + regs[rt]; break;
 case 0x22: regs[rd] = regs[rs] - regs[rt]; break;
}}}}}}
```

This is in fact the code that would be executed when simulating the ADD instruction. Notice that the operation code `opc` is set to a constant because this is the semantics code for the addition instruction and the element `opc_add` returns the value 0x20, which is used as the operation code.

The approach chosen to represent unknown operand values is similar to symbolic execution (e.g. [13]). The `codasip_regopindex` function call represents the register index value (symbol value) in a C-compiler compatible fashion. The `codasip_regopindex` argument is the operand index and the mapping to operands is stored in the instruction syntax representation:

```
"ADD" gpreg(0) "," gpreg(1) "," gpreg(2)
```

Such a C code and syntax are generated for each instruction as a function. However, there is still a lot to be done in order to simplify the semantics to get it to a form suitable for C compiler generation. Such representation must be transformable into the instruction selection patterns defined as trees; for example, the ADD instruction pattern must be as follows.

```
(set gpreg:$op0,
  (i32 (add
         (i32 gpreg:$op2),
         (i32 gpreg:$op1)
  ))
)
```

The same pattern is shown graphically like a tree in Figure 3.3.

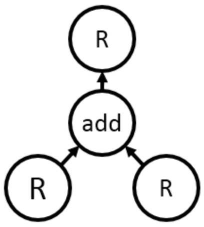


Figure 3.3: Desired form of instruction semantics representation of and addition instruction

We will describe the transformation and simplification process that is able to transform most instructions into this tree-like (or DAG-like) representation in the following sections.

Memory and Register Information Generation

To be able to perform the simplification, we must define the processor resources that the instructions use. You can notice that in the example 3.1.1 an array regs is used. Instructions can also access the memory and read from or write into the program counter register. We have several registers and a memory defined in the following CodAL example:

```
program_counter bit[32] pc;

arch register bit[32] regs[32];
register bit[32] tmp_reg;

memory bit[32] mem {
  .endianess = big,
  .lau = 8 // byte size
};
```

In the instruction semantics model, we must distinguish between architectural and non-architectural registers, program counter, and memory accesses. Architectural register accesses must be represented in the instruction semantics as register accesses. Nonarchitectural registers must not be present in the semantics at all, because they are used to store temporary data within one instruction execution. Nonarchitectural register accesses must not have any effect on the architectural state of the processor. A read of a program counter represents the retrieval of the current instruction address; this behavior is used in calls and in relative jumps. A write to a program counter is a jump. Finally, memory accesses must be represented as loads and stores.

To correctly compile the functions generated for each instruction such as the one in example 3.1.1, the definiton of registers and memories is also generated as a C code.

```
volatile  uint32 pc;
volatile  uint32 regs[32];
volatile  uint32 mem[2048];
volatile  uint24 mem___sb3[2048];
volatile  uint16 mem___sb2[2048];
volatile  uint8  mem___sb1[2048];
uint32           tmp_reg;
```

You can see here the difference between the architectural register file `regs` and the non-architectural register `tmp_reg`. The architectural register file is generated as a volatile global variable and the `tmp_reg` is a standard global variable. A load from a volatile variable must be kept by the C compiler while the non-volatile variable load can be optimized away.

Another feature is the *subblock* memory access. In a CodAL model, a 32-bit memory with 8-bit bytes can be accessed either as a whole word (32 bits), a 24-bit sized value, a halfword (16 bits), or a byte (8 bits). To correctly represent each of these possible accesses, the memory representation in the C code is replicated to allow each of these different access methods. Memories must be always volatile because memory access is an important instruction operation which cannot be omitted; for example, for some architectures a load outside the permitted range causes an exception, so this behavior must be kept in the instruction semantics. The program counter is also generated as a volatile variable.

To know which variable was a register, program counter or memory, an auxiliary file is also generated that holds this information.

3.1.2 Processing and Optimizing Instruction Semantics

In the previous section, the generation of unprocessed instruction semantics codes is described. Now we take a look at how such a code can be processed and simplified. The task here is to get the simplest possible representation of an instruction behavior or semantics. For example, the ADD instruction code in 3.1.1 must be simplified to a simple dataflow graph such as the one shown in Figure 3.3.

Now, the question is how we can perform this transformation. One way is to write a program that will process the input C code (there is already a C language parser in the CodAL language) and implement some simpler optimizations like constant propagation and dead code elimination, and this would suffice for our example instruction. However, much more complex instructions can be present in the instruction code; for example, it is usual that function calls are used. This would mean implementing a function inlining pass and many others, and also trying to match the code transformations made in LLVM so that the resulting patterns match the patterns produced by LLVM. This match is necessary for seamless instruction selection.

Another option is to use an already existing C language frontend, optimizer, and also backend and in fact *lower* the semantics representation to the same level as the one used for instruction selector in a compiler backend, as shown in Figure 3.4.

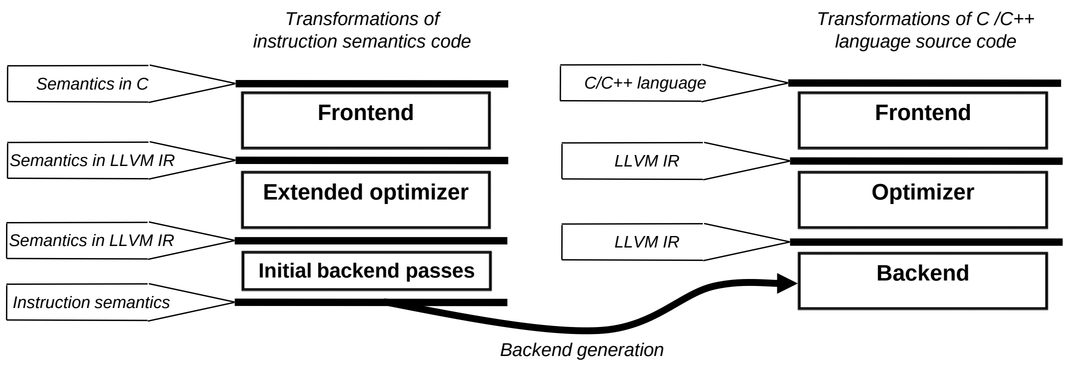


Figure 3.4: Lowering of instruction semantics representation to match the compiled code representation at instruction selection level

The semantics extractor consists of three tools that generate and transform the instruction semantics, they are: *semextr*, *opt-semextr*, and *llc-semextr*. In section 3.1.1, we already described the tool *semextr*. In the following sections the tools *opt-semextr* and *llc-semextr* will be described.

Parsing and LLVM IR Generation

Parsing and LLVM IR generation are the first part of processing, where the input semantics code is parsed by a compiler frontend *clang* and unoptimized intermediate representation is generated.

One important part here is the function attribute specification. To allow a correct semantics analysis in the frontend, auxiliary fuctions such as the `codasip_regopindex` introduced in section 3.1.1 must be declared (the compiler must know the function interface). However, as the function represents an unknown symbol from the symbolic execution point of view, the function must not be defined (the compiler must not know the function body). The following LLVM optimization passes must know what to expect from such a function to optimize the code as much as possible. Therefore, the `codasip_regop` function and similar ones must be declared with the C language attribute `const` [30] specifying that the result of the function depends only on its input arguments an does not access the memory. If the C optimizer were not unaware of such a function constraint, a function call to `codasip_regop` would be considered a boundary for optimizations such as constant propagation.

Another issue specific to semantics extraction is the use of data types with non-standard bitwidths. For example, an instruction can have a 5-bit immediate operand that must be modeled correctly as a 5-bit constant value. There is no way how to declare such 5-bit variables in C, so the author had to extend the *clang* frontend with a `bitwidth` attribute as shown in the following code.

```
typedef int int5 __attribute__((bit_width(5)));
```

With these extensions, the *clang* can parse the code and generate LLVM IR.

```
define void @instr__opc_add__gpreg__gpreg__gpreg__() nounwind {
entry:
  // Allocate space for local variables on stack.
  %opc = alloca i32, align 4
  %rd = alloca i32, align 4
  %rs = alloca i32, align 4
  %rt = alloca i32, align 4
  store i32 32, i32* %opc, align 4

  // These are the codasip_regopindex calls representing symbolic
  // register operand indexes.
  %call = call i32 @codasip_regopindex(i32 0, i32 0) nounwind
     readnone
  store i32 %call, i32* %rd, align 4
  %call1 = call i32 @codasip_regopindex(i32 0, i32 1) nounwind
     readnone
  store i32 %call1, i32* %rs, align 4
  %call2 = call i32 @codasip_regopindex(i32 0, i32 2) nounwind
     readnone
  store i32 %call2, i32* %rt, align 4

  // Load first input register value.
  %0 = load i32* %rs, align 4
  %arrayidx = getelementptr inbounds [32 x i32]* @regs, i32 0, i32
      %0
  %1 = load volatile i32* %arrayidx, align 4

  // Load second input register value.
  %2 = load i32* %rt, align 4
  %arrayidx3 = getelementptr inbounds [32 x i32]* @regs, i32 0, i32
      %2
  %3 = load volatile i32* %arrayidx3, align 4

  // Do the addition.
  %add = add i32 %1, %3

  // Store the resulting register value, the first instruction
  // %4 = load loads the destination register index.
  %4 = load i32* %rd, align 4
  %arrayidx4 = getelementptr inbounds [32 x i32]* @regs, i32 0, i32
      %4
  store volatile i32 %add, i32* %arrayidx4, align 4

  ret void
}
```

The *clang* frontend did already some optimizations like constant propagation and dead code elimination because the original switch from C code 3.1.1 is not present in this representation anymore. You can notice that there is an `add` operation that says what this operation really does. But there are still many other operations that need to be optimized away.

Semantics Code Optimization

In this section, the next step of semantics extraction is described. This step is done by the tool *opt-semextr*, which runs over optimizations and over 30 custom transformations in a required order. Subsequent to *opt-semextr*, the LLVM IR representation is already really very close to the form needed by compiler backend generation. When *opt-semextr* finishes,the tool *llc-semextr* is run, which performs the final transformations. We will describe the most important passes of the tool *opt-semextr* in the following text.

Inlining In many cases, an auxiliary function is called from the instruction semantics code. Such calls have usually some constant arguments (such as opcode) as shown in a slightly modified example from 3.1.1.

```
int32 compute_arithm_operation(int opc, int32 src1, int32 src2)
{
  switch (opc)
  {
    case 0x20: return src1 + src2;
    case 0x22: return src1 - src2;
  }
}
```

```
void instr()
{
  const int opc = 0x20;
  const int rd = codasip_regopindex(0);
  const int rs = codasip_regopindex(1);
  const int rt = codasip_regopindex(2);
  regs[rd] = compute_arithm_operation(opc, regs[rs], regs[rt]);
}
```

To maximally simplify the instruction semantics code, we must inline the called function. For each such auxiliary function that is not marked specifically as no-inline, an attribute is set to do the inlining. Then a standard LLVM inliner pass is run.

Handle user functions Often repeated calculations in instruction semantics such as setting of flags may be described as user functions in the extracted semantics. This can greatly improve the readability of the resulting instruction semantics and make it smaller, because for architectures like Intel 386 the resulting semantics file is huge. The pass *Handle user functions* removes unused functions that represent neither instructions nor user functions, and stores information about the names of user functions for later use.

Optimization -O3 All optimization passes that are run with -O3 option for the LLVM optimizer are run in this phase. The most important optimizations are here: constant propagation, constant expression evaluation, control-flow graph simplification, and dead code elimination. Addresses of direct register accesses (accesses to a register that is specified in instruction operations code such as carry flag or to the return address register) are propagated directly to the memory access location which is used by the following pass. Together with using *clang* as the frontend, using these optimization passes is one of the biggest advantages of using LLVM to generate itself. Many man-years of work were spent on implementing these passes, and this way they can be reused exactly as they are.

Direct register rewriting All accesses to registers with constant register addresses are replaced at this point with special intrinsic instructions. These intrinsics specify register reads or register writes. Constant register addresses appear in functions like `CALL`, where the return address is stored into an implicit return address register (for example `R31`). Another case handled here is access to flags. Physical (implicit) register accesses are encoded into the instruction opcode, whereas for register operand accesses (logical registers) the address of the register operand is encoded separately in the instruction binary coding. All implicit register accesses are processed by this pass.

In the following example, we have a store to a memory location that represents the register `R31` (global array called `regs`).

```
store volatile i32 %0,
  i32* getelementptr inbounds ([32 x i32]* @regs, i32 0, i32 31)
```

We rewrite it to an intrinsic function `@llvm.regwrite.anyint`, where the arguments specify the register file (resource `regs` has index 5), the register index (31), the value to be stored (variable %0), and finally a condition for this write. Allowing conditional writing to a register allows in the latter passes better instruction reordering that leads to fewer basic blocks (and simpler semantics).

```
call void @llvm.regwrite.anyint.i32(i32 5, i32 31, i32 %0, i1 true)
```

Later, we will rewrite the `codasip_regopindex()` calls with a constant to enable additional optimizations. We need to process physical registers at this stage because after replacing `codasip_regopindex()` with a constant, it will be impossible to distinguish which register access was a physical register access and which was a register operand access.

Propagate register index addresses This analysis finds all occurrences of calls to `codasip_regopindex` and follows the use-def chain [81] datapath to a place where the register index value is used as a register operand address. The datapath analysis is needed to support indexed register operands. An example of indexed register operand is when some instruction accesses a register pair `rN` and `rN+1`, but only the value N is encoded as a register operand in the instruction binary coding.

The operations `add` and `sub` with a second constant operand are removed and the offset is remembered. The operations truncate and and on the datapath from the register operand address to the register access can be ignored only if the result cannot change the operand address value.

Finally, the original `codasip_regopindex` calls are replaced with a constant address that was selected from allowed register operand addresses (e.g. when the registers `R1-R31` form a register class, then one value from 1 to 31 is chosen). Each different register operand gets a different constant index that is remembered for the following pass called Register operand rewriting.

Constant registers are also handled in this way. For example, in the MIPS architecture the general purpose register (GPR) with address 0 is always zero. In the CodAL description, this behavior is described using a condition that compares the register address with the constant 0. If it is equal, then a write does nothing and a read returns 0. Every access to a GPR in the MIPS CodAL model uses the following macros, which define this zero-register functionality.

```
#define RWRITE(reg, val) { if ((reg) != 0) gpregs[(reg)] = (val); }
#define RREAD(reg) ((((reg) != 0) ? gpregs[(reg)] : 0))
```

Different instructions are generated by the tool *semextr* for each combination of register usage. In MIPS, we have GPR operands that can be either `R0` (one register), or `R[1-31]` (register class). This is defined in CodAL as:

```
arch register bit[32] gpregs[32];

set gpr_std represents gpregs =
    gpr1 , gpr2, ..., gpr31;

set gpr = gpr_std, gpr0;
```

For the MIPS instruction `ADD` with three register operands defined by the set `gpr`, there are eight versions of the `ADD` instruction: 1) `ADD R[1-31], R[1-31], R[1-31]`, 2) `ADD R[1-31], R[1-31], R0`, 3) `ADD R[1-31], R0, R[1-31]`, etc.

The replacement of `codasip_regopindex` with a constant allows optimizing away the conditions in the `RWRITE` and `RREAD` macros. For access to register operands (R[1-31]), this must be done differently. An intrinsic instruction, `propagatedregopindex`, is created directly before the register access that uses this address. One operand of the operation `propagatedregopindex` contains the calculated offset for indexed access (from the `add` and `sub` operations on the datapath). We cannot let the optimizer propagate the assigned address because the addition and subtraction operations (allowed for indexed register access) would change the constant assigned address. Therefore it would not be possible to identify which address was assigned to which operand. The next important step after this transformation is to run the optimizer.

Optimization -O3 All -O3 optimization passes are run again. The main reason is to remove conditions for constant register accesses as shown in 3.1.2 since the original `codasip_-regopindex` calls were replaced with a constant address.

Register operand rewriting Now, when accesses to constant registers have been handled, only the `propagatedregopindex` operation represents a register operand address. We replace the following two pairs of operations, `propagatedregopindex` and load, and `propagatedregopindex` and store, with the intrinsic operations `regopread` and `regopwrite`. The new operations then specify register operand accesses. The operations `regopread` and `regopwrite` have the register class index and operand index as operands.

Here is an example of the code created by the pass *Propagate register index addresses*. The value of `@llvm.propagatedregopindex` is used to address the `regs` global array and then there is a load from the `regs` array.

```
%0 = tail call i32 @llvm.propagatedregopindex.i32(i32 0,i32 1,i32 0)
%arrayidx = getelementptr inbounds [32 x i32]* @regs, i32 0, i32 %0
%1 = load volatile i32* %arrayidx, align 4
```

All these three operations are replaced by one operation representing the register operand read. The arguments 0, and 1 specify the register class `gpregs` and the operand with index 1, respectively.

```
%0 = call i32 @llvm.regopread.anyint.i32(i32 0, i32 1)
```

Remove unused register read operations This is a simple pass that removes all register read operations because reading a register does not change architectural state. Unlike a memory access, a register read cannot cause an exception, so it can safely be removed from the instruction semantics.

Replace Codasip builtins Some operations such as floating-point comparisons are hard to describe precisely in the C language. For example, it is not clear whether the comparison $f1 > f2$ means ordered or unordered comparison [100], where in unordered comparison the result is true if one of the operands is NaN (not a number). To precisely describe this behavior, special functions were introduced in the CodAL language. One example is `uint1 codasip_fcmp_oeq_float(float, float)`, which explicitly specifies ordered equal comparison. This operation is then replaced by the LLVM IR operation `fcmp oeq`.

Another case handled by this pass concerns bitcasts from floating point to integer that the user explicitly specified as bitcasts and not register accesses. Bitcast is an operation that copies bits in a variable as they are, no conversion is applied. These builtins are replaced with LLVM IR `bitcast` operations, so that they can be optimized later by standard optimization passes.

Memory access rewriting This pass is similar to the *Direct register rewriting pass*. It finds memory accesses and replaces them with intrinsics that specify these accesses.

Memory accesses are modeled originally with a volatile global array access

```
%arrayidx4 = getelementptr inbounds[2048 x i32]*@mem,i32 0,i32 %add
store volatile i32 %1, i32* %arrayidx4, align 4
```

This access is replaced with a memory access intrinsic.

```
call void @llvm.memwrite.anyint.i32.i32(i32 %add, i32 0, i32 %1)
```

Floating point and vector register rewriting Floating point and vector register accesses are specified as a pair of floating point or vector value bitcast and an integer register access. The registers are always represented as integer arrays, but to simplify the resulting semantics description the pairs of bitcasts and integer register accesses are replaced with special intrinsic. Both register operands and fixed registers are handled here.

Switch lowering Some instructions may be described by switches. Also the optimizer may decide that sequence of conditional executions (with `if ... else if ...`) will be implemented more efficiently with a switch. This complicates the subsequent passes and, moreover, switches are not allowed in the resulting semantics. This standard LLVM pass transforms switches into sequences of conditional executions.

Compacting short evaluation and phi transformation This pass consists of two parts. The goal is to eliminate as many conditional branches from the description as possible. This is done by eliminating the `phi` operations [101] and replacing them with `select` operations. This pass is able to transform very complex control flow graphs into just one basic block.

If the instruction semantics is defined with just one basic block that produces one result, it can be always transformed into instruction selection pattern. Then this instruction can be used automatically, when the same pattern appears in the input code of the C compiler.

We will describe this pass in more detail. In the *phi transformation* several control flow patterns are searched. An example of such a pattern is shown in Figure 3.5.

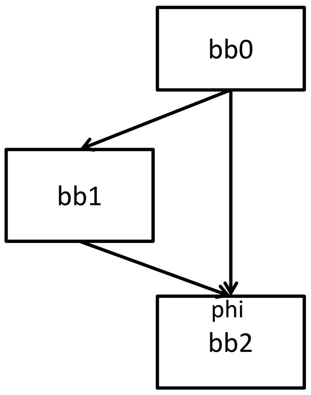


Figure 3.5: Control flow graph pattern to be transformed by phi transformation

The basic block `bb1` must not have any side effect such as memory access or register write. It is then safe to merge all the basic blocks into one, as shown in the following example of the instruction `dst = logor src1, src2, imm`, which performs the operation `dst = (src1 || src2) ? imm : 0`. It first computes the logical *or* on source operands and then, if the result is true, the imm operand is stored into the destination register. If the result is false, then zero is stored.

You can also see in the example how the shortened boolean expression evaluation works. In bb0, the operand src1 is first evaluated and if it is true (not equal to zero), the the code jumps to bb2, where the imm operand is stored into dst. Only in the case when src1 is false, is the src2 operand evaluated.

```
define void @i_test_logor() {
bb0:
  // Read immediate operand.
  %call3 = tail call signext i5 @codasip_immread_int5(i32 3)
  %imm = sext i5 %call3 to i32
  // Read operand src1.
  %0 = call i32 @llvm.regopread.anyint.i32(i32 0, i32 1)
  // Shortened evaluation for the case when src1 == false.
  %tobool62.i = icmp eq i32 %0, 0
  // Go to bb2 if src1 == true.
  br i1 %tobool62.i, label %bb1, label %bb2
bb1:
  // Read operand src2.
  %1 = tail call i32 @llvm.regopread.anyint.i32(i32 0, i32 2)
  // Test whether src2 != false.
  %tobool63.i = icmp ne i32 %1, 0
  // Compute phitmp = (src2 != false) ? imm : 0.
  %phitmp = select i1 %tobool63.i, i32 %conv, i32 0
  // Go to bb2.
  br label %bb2
bb2:
  // If we arrived from bb0, use imm, else
  // if we arrived from bb1, use phitmp.
  %2 = phi i32 [ %imm, %bb0 ], [ %phitmp, %bb1 ]
  // Store result of phi into destination register.
  call void @llvm.regopwrite.anyint.i32(i32 0, i32 0, i32 %2, i1
     true)
  ret void
}
```

This is the code we get after the phi transformation. The semantics of the code stays the same, but we have eliminated all the unnecessary conditional jumps. Also, all of the instruction semantics is in just one basic block.

```
define void @i_test_logor() {
sw.bb61.i:
  // Read immediate operand.
  %call3 = call signext i5 @codasip_immread_int5(i32 3)
  %imm = sext i5 %call3 to i32
  // Read operand src1.
  %0 = call i32 @llvm.regopread.anyint.i32(i32 0, i32 1)
  // Evaluate first part of condition: cond1 = (src1 == false).
  %cond1 = icmp eq i32 %0, 0
  // Read operand src2.
  %1 = call i32 @llvm.regopread.anyint.i32(i32 0, i32 2)
  // Evaluate second part of condition: cond2 = (src2 == false).
  %cond2 = icmp ne i32 %1, 0
  // Compute tmp1 = (cond2) ? imm : 0.
  %tmp1 = select i1 %cond2, i32 %imm, i32 0
  // Compute tmp2 = (cond1) ? tmp1 : imm.
  %tmp2 = select i1 %cond1 , i32 %tmp1 , i32 %imm
  // Store tmp2 into dst.
  call void @llvm.regopwrite.anyint.i32(i32 0, i32 0, i32 %tmp2 , i1
      true)
  ret void
}
```

This phi transformation is coupled with the shortened evaluation of boolean expressions. The purpose of this transformation is to merge the basic blocks by computing a conditional jump condition, using logical operations instead of computing it with conditional jumps. An example of the input control flow graph is given in Figure 3.6. A conditional jump from `bb0` depends on the value `a`, and a conditional jump from `bb1` depends on the value `b`. Now, when there is no side effect in `bb1`, the basic blocks `bb0` and `bb1` can be merged and the conditional jump then depends on the result of logical and of `a`and `b`, as shown in Figure 3.7. Also the control flow graph (CFG) is simplified with this type of transformation.

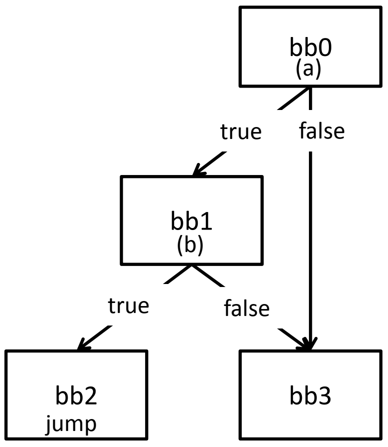


Figure 3.6: Input CFG to compact short evaluation transformation.

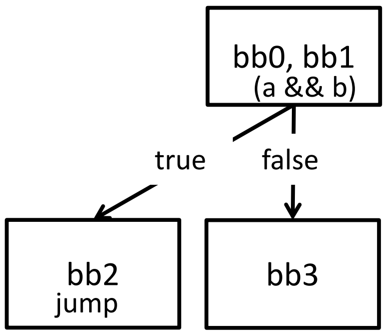


Figure 3.7: Resulting CFG with pre-computed condition.

The compaction of shortened evaluation is essential for flag-based architectures, where a jump condition may depend on a logical operation over flags; for example, the condition for an *unsigned greater than* comparison is calculated as *not(carry) && not(zero)*. If this computation were spread over multiple basic blocks, the detection in the compiler generator would be much more complicated than analyzing the output of the Compact shortened evaluation pass.

The phi transformation and the compacting of shortened evaluation are repeated iteratively, since changes made by one transformation may allow the next transformation to change the code further. So these transformations are run until there is nothing left to be changed.

Semantics finalization This pass was originally the last in the semantics extraction. It prints the instruction semantics into a file on the abstraction level of LLVM IR. Our example instruction for addition then looks like this.

```
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;  // Get value of 1st register operand
  %u1 = i32 gpreg_2;  // Get value of 2nd register operand
  %add = add(%u1, %u0); // Add the 2 values
  gpreg_0 = %add;     // Put the result into the dest. register
  ,
  // Syntax
  "ADD" gpreg_0~"," gpreg_1~"," gpreg_2,
  // Binary coding
  0b000100 gpreg_0[4,0] gpreg_1[4,0] gpreg_2[4,0] 0b00000000000
```

We can rewrite the semantics code into an expression and now we can see that the form is finally the tree-like representation we wanted to obtain, as shown in Figure 3.3.

```
  gpreg_0 = add(gpreg_1, gpreg_2)
```

The output containing such a description of all instructions is stored into a file called `simulator_semantics.sem` by this pass. This representation is then used to generate QEMU-based simulator, decompiler, and for other purposes, as described in attachment E.

For the C compiler generation additional passes were added. They deal mainly with conditional writes that are not supported in the `simulator_semantics.sem` file. These passes prepare the code for *llc-semextr*, which, due to LLVM limitations, discards all instructions whose semantics cannot be converted to a single basic block. Other tools that use the semantics generated by the Semantics finalization pass do not have such limitations and this is the main reason for having two files that define the instruction semantics.

LLC extractor The LLC extractor is a modified LLVM backend that performs the lowering and legalization (see 2.2.3) passes, which may reorder several arithmetic operations. They also translate the LLVM IR into the Selection DAG representation, which uses SD Nodes instead of LLVM IR operations.

The same transformations in lowering and legalization are applied to the C or C++ code being compiled by the LLVM backend. Using the *llc-semextr* allows applying the same transformations to the instruction semantics. When the instruction selection patterns are generated from the semantics, they better match the form of the code being compiled by the generated backend.

The LLC extractor generates the file `compiler_semantics.sem`, which has the same form as the file `simulator_semantics.sem`. Only several operations are different, this difference follows from the differences between the LLVM IR and Schedule DAG (SD Nodes) representations.

3.1.3 Design Principles of Semantics Extractor

The Semantics extractor tool has been under development for several years and has been extended gradually to handle more and more architectures. This section lists several design principles that were employed during the implementation.

No Stopping On Problematic Instructions

The user needs to get a working C/C++ compiler as soon as possible. Then they can focus on optimizations and on instructions that were not correctly defined or are not supported by the Semantics extractor in their current form.

Because instructions with unexpected or malformed codes can be generated, the Semantics extractor must stop only at fatal errors. For most problems, only a warning is printed and the instruction with malformed code is invalidated. The reason for invalidation is stored as a comment on the removed instruction.

Example of an instruction that can be defined in CodAL and cannot be used automatically by the C compiler is an instruction that is described using a loop. For example, it may be convenient to describe a bit counting instruction using a loop with a non-constant iteration count.

However, the instruction semantics format cannot describe loops. There are no such high-level constructs in the format, and labels are not supported either, so, instructions with loops are not supported. An instruction with a loop cannot be used automatically by the compiler, so this does not represent a limitation. Instead of printing an error, the instruction semantics is removed and the instruction is marked as undefined. Then the user can go through the semantics file, see why the instruction was removed and fix it in the original CodAL file.

Simple Passes

Another principle that has proved to be very beneficial is splitting the transformations into many simple passes. Even if some passes could be merged into one pass that would perform the transformation faster, creating a new pass was preferred.

The LLVM pass manager has good support for specifying the order of executed passes. This way can be the pass execution reordered and this also allows executing the same passes several times over an extracted code in different phases.

Printing Intermediate Code

The pass manager used in *opt-semextr* always prints the intermediate representation (LLVM IR) before a pass is executed. Debug printouts can be disabled, but by default the printouts are enabled even for release builds.

This greatly simplifies debugging the Semantics extractor, since all intermediate semantics forms are available and the user can see what is happening to a problematic instruction. Developers can quickly identify a problematic pass that wrongly transforms an instruction.

Mapping to the Original CodAL Code

For complex models, it can be hard to find a problem in the model where an instruction is wrongly described. Due to all optimizations that are run over the original C code, it is practically impossible to map operations from instruction semantics onto the code from the CodAL `semantic` sections. As an example, we may need to report that an out-of-range access to a register file was originally defined on a certain line in the CodAL model. For the -O3 optimization level, the resulting code is completely different from the input. Even the original memory access can be replaced with a different instruction, so mapping from an LLVM IR operation to the original CodAL code cannot be maintained. However, the extractor knows what elements an instruction was constructed from and this can help the user to find the problematic place. A tree-like structure containing information about elements is used to construct this instruction, for example: `el:i_2_reg_operands(el:opc_-mov, el:gpreg, el:gpreg)`. The Semantics extractor and also the Backend generator then use this information to report the problematic instruction:

```
Warning (SE605) for instruction i_2_reg_operands__opc_mov__gpreg__gpreg__:
Index for addressing register operand from registers class 'gpreg' is out
of bounds. ...
Instruction was constructed from:
  el:i_2_reg_operands(el:opc_mov, el:gpreg, el:gpreg)
```

The `el:i_2_reg_operands`, `el:opc_mov`, etc. are converted by the graphical user interface into links that point to definitions of elements in the CodAL model. This way the user can find very quickly the parts of a model that may cause the problem. The warning with ID `SE605` is also a reference to help with a more detailed explanation of the message.

3.1.4 Semantics Extraction Results

The purpose of the Semantics extractor is to prepare an instruction definition for the C compiler generation. The 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.

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

The results for diverse architectures are presented in Table 3.1. These results were obtained for the 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). The size in bytes contains the resulting size of the `compiler_semantics.sem` file. To show the relative size needed per one instruction, the Bytes per instruction column contains the 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, which has on average more complex instructions). Architectures whose arithmetic instructions set flags such as carry, overflow, etc., have a higher average size per instruction, because setting these flags is included in the instruction semantics.

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

The high runtime for the Vix architecture is partly given by the binary coding generation in the tool *semextr*, because its instruction set coding is not well suited for decoder generation used to obtain the binary coding of each instruction. The high runtime for the Intel 386 architecture is caused by the model description. For Intel 386, the behavior of all instructions is described by several large functions and the tool *semextr* generates such a C code that contains calls to these large functions that are later inlined. Then the tool opt*opt-semextr* must optimize many lines of code for each instruction.

Architecture	**Instructions**	**Size in bytes**	**Bytes per 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

ADOP [56] is a 16-bit architecture developed at Czech Technical University in Prague; ARM7 cc is a simplified version of the ARM v7 arch. [4]; AVR32 [7] is a 32-bit arch. by Atmel; Codix RISC [41] is a 32-bit extensible arch. (described below, in section 3.3.3); Codix RISC VLIW is a VLIW version of the Codix RISC; Codix Experimental is Codix RISC extended with SIMD instructions; Codix Stream [40] is a 16-bit DSP architecture (described in 3.3.2); Codix uRISC [43] is a 32-bit minimalistic architecture (described in 3.3.1); Infineon Tricore [49] is a 32-bit DSP from Infineon; Intel 386 [51] is a full version of the Intel 386 including the 387 floating point coprocessor; Microblaze [119] is a 32-bit soft-processor from Xilinx; MIPS basic is a simplified version of a 32-bit MIPS [108] arch. without floating point instructions; MIPS is a full model of MIPS Release 1 arch. with DSP and floating point instructions; Open RISC [23] is an open-source 32-bit arch. from Open Cores; Power PC [47] is a 32-bit arch. from Apple, IBM, and Motorola; Simple Flag and Simple Flag

Float are experimental 32-bit architectures that use flags as conditions for branches; Vix [42] is a 32-bit architecture that was planned to be used as a base for a new VLIW processor.

The instruction semantics format was also successfully used for fast simulator generation [77], in a reverse compilation tool [58], and in a processor verification tool [15]. Semantics extraction was published by the author in [45], and in [46].

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. No inconsistencies can occur such as in the LISA ADL approach, where there are two 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 operation ordering is exactly the same as that which appears during compilation. This assures that instruction selection patterns generated from the instruction semantics match the compiled code.

Using strong optimizations and additional transformations, instructions originally described with complex C code are optimized 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 we have just described converts the CodAL model into a representation suitable for C compiler generation. From this description a C compiler backend is generated. 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 preparing the tests, being in charge of the implementation work, and placing priorities on new features, optimizations, and stability. Most of the implementation work and internal components design was carried out by Jan Hranáč. Testing together with its automation was done by Luděk Dolíhal, who was also the main quality engineer for the compiler generator. Additional optimizations were done as part of numerous bachelor and master theses led by the author and these optimizations and extensions are briefly described at the end of this section (3.2.3).

3.2.1 Overal Compiler Generator Design

As the basis for the generated C compiler the author chose LLVM [59], reasons to choose LLVM were explained in section 2.2.8. To shortly recapitulate: cleaner C++ implementation than GCC, good documentation compared to other compilers, strong user base, and a high likelihood of wide acceptance of LLVM for embedded systems.

The author chose LLVM as the basis for the generated C compiler, reasons for choosing LLVM were explained in section 2.2.8. To briefly recapitulate: cleaner C++ implementation than GCC, good documentation compared to other compilers, strong user base, and a high likelihood of wide acceptance of LLVM for embedded systems.

The structure of the Backend generator is shown in Figure 3.8. The Backend generator has the 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. The user may provide additional inputs when they need 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 that the Backend generator can recognize such an 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 lack several comparison types, so with these equality rules the user can define which instructions should be used, for example 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 the automatically generated instruction schedule. The standard LLVM definition for the LLVM `tablegen` tool is used and this allows, for example, specifying that a load from the memory has a latency of 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 an operation.

Tablegen and C++ file templates are another input for the Backend generator. This input can be modified by a Backend generator developer and should not be usually modified by the user. They are written in a PHP-like code, which is then processed by the Templates generator. A simple example is the file `CallingConv.td.tp`, which is used to generate the file `CallingConv.td` containing the calling convention definition for the LLVM backend.

```
def CC_Codasip_Gen: CallingConv<[
  $$( PrintCallConv(OUT); $$)
]>;

def RetCC_Codasip_Gen: CallingConv<[
  $$( PrintRetConv(OUT); $$)
]>;

include "CodasipCustomCallC.td"
```

First, the *Instruction set analyzer* processes its inputs and stores them in structures for the *LLVM code generator*. The *Templates generator* is then started. It prints the text from

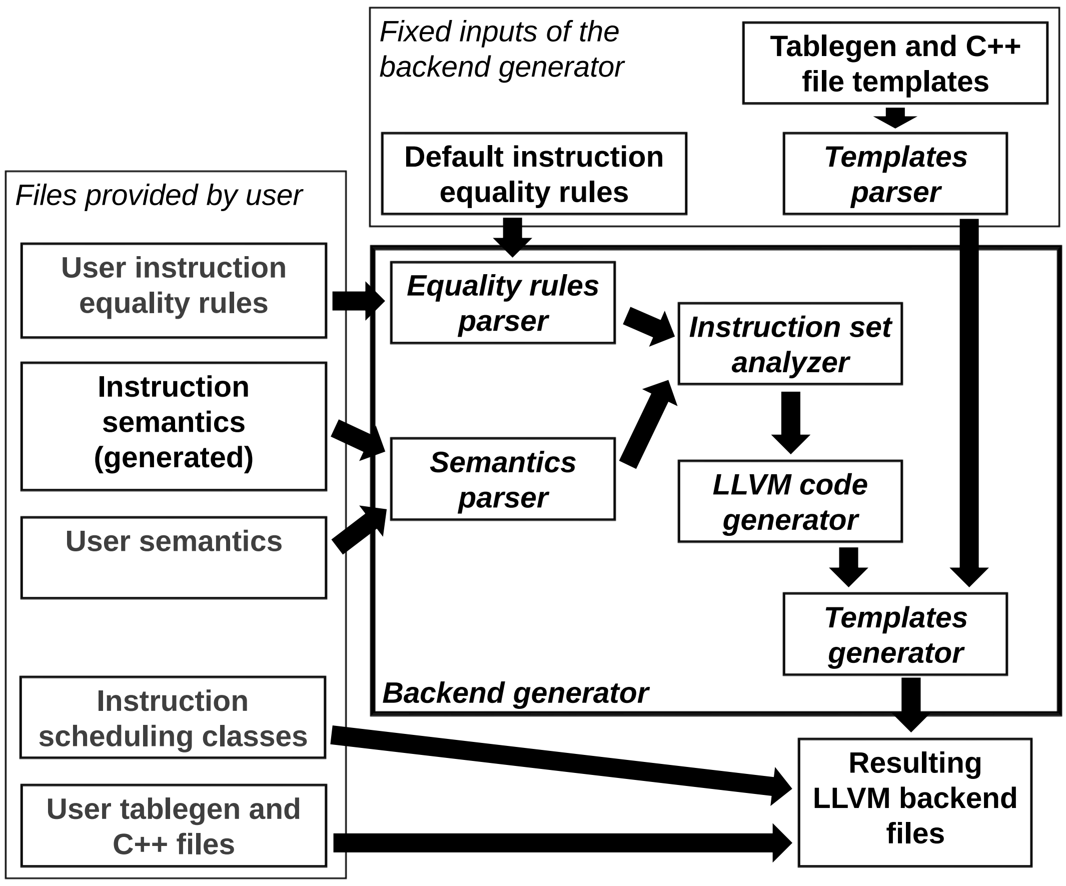

Fixed inputs of the backend generator
Tablegen and C++ file templates
Default instruction equality rules
Templates parser
Files provided by user
User instruction equality rules
Equality rules parser
Instruction set analyzer
Instruction semantics (generated)
Semantics parser
LLVM code generator
User semantics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

Figure 3.8: Inputs and main components of the backend generator

template files into particular files and once it encounters a code enclosed in `$$( ... $$)`, it executes this code (this is similar to HTML code generation with PHP). In our example, the LLVM code generator function `PrintCallConv` is called. It outputs the calling convention definition. An example of a full calling convention definition was shown in section 2.2.3. This way every input template from Tablegen and C++ file templates is processed and the *Resulting LLVM backend files* are created. The files from *Instruction scheduling classes* and *User tablegen and C++ files* then replace some of the automatically generated files. Then the LLVM backend is built into an executable binary.

3.2.2 Backend Source Files Generation

The process of LLVM IR translation with a backend was described in section 2.2.3. LLVM needs several files that define the target architecture and these files usually need to be written by the compiler developer, as was also shown in section 2.2.3. The most important files defining the target architecture that are automatically generated by the backend are the following:

ISelLowering.td contains the definition for the *lowering* and *legalization* passes (2.2.3). The *Instruction set analyzer* must find out which operations and data types are supported by the target, and generate information on whether a particular operation should be kept for instruction selection or must be transformed into other supported instructions.

RegisterInfo.td defines available registers and data types supported by the registers. Registers and register classes are present in the generated *Instruction semantics*, data types supported by these registers are determined from instructions that use these registers. This information is used by the *legalization* and *register allocator* passes.

InstrInfo.td is the main source of information for the instruction selector, where for each instruction its instruction selection pattern (see 2.2.3) and also its assembly syntax are defined. Each instruction also has flags, showing whether it is a branch, load or store, and if the instruction may have any side effects. Details about each instruction are obtained by an analysis of the instruction semantics. Assembly syntax is then used by the assembly printer pass. An example of a load instruction definition that is used in `InstrInfo.td` is displayed here.

```
def i_load_store__opc_load__gpreg__am_base_offset__gpreg__simm16__:
  // This instruction has for scheduling assigned a
  // microarchitecture class 'loads'.
  CodasipMicroClass_loads<
  // Instruction operands.
  (outs gpreg:$op0), (ins gpreg:$op1, i32imm:$op2)
>
{
let AsmString = "LOAD $op0, $op1 + $op2";
// Instruction selection pattern, the sext16To32imm
// operation is a constraint on immediate operand
// that the operand must be expressible with a 16-bit signed
// value.
let Pattern = [
  (set gpreg:$op0,
    (i32 (load (i32
      (add
        (i32 gpreg:$op1),
        (i32 sext16To32imm:$op2)
      )
  ))))
)];
// Size is determined from binary instruction coding.
let Size = 4;
// Instruction may load from memory, but does not store.
let mayLoad = 1;
let mayStore = 0;
}
```

Patterns.td is an additional source of information for the instruction selector. The *lowering* and *legalization passes* passes are very general for arithmetic operations, but are very weak for comparisons. These passes are not able to transform the Selection DAG in a way that only supported comparisons are present. The *Instruction set analyzer* needs to find operations for comparisons, selects, and conditional branches, and for moving a pointer-sized constant to a register. When certain operations (such as `branch if greater than`) are not found, the *Instruction set analyzer* looks into the *Default instruction equality rules* or the *User instruction equality rules* to find equivalencies that can be implemented by available instructions.

For instance, the `branch if greater than` (`bgt src1, src2`) operation for integer data types can be implemented by branch if lower or equal with its operands switched (`ble src2, src1`) or by an if greater than comparison followed by a conditional `branch if not zero` (`tmp = cmpgt src1, src2; bnz tmp`).

The *Instruction set analyzer* an equivalency with a minimal count of needed instructions and generates equivalency patterns into the `Patterns.td` file. If no suitable equivalency is found, a severe warning during backend generation is printed. Then, when the generated backend compiles an application that needs a missing operation the backend fails to compile the application with a *Could not select error* from the instruction selector.

GenFrameLowering.cpp contains the C++ code, which is called by the LLVM backend infrastructure and emits the function prologue and epilogue. The *Instruction set analyzer* needs to find instructions that subtract an immediate from the stack pointer register, and also load and store instructions that compute address by adding a register and an immediate.

GenRegisterInfo.cpp finally contains the C++ code, which is called by the register allocator when a register is spilled. For each register class, there must be instructions that can store a register to the stack and also restore it.

These are the most important files generated by the Backend generator. The C++ files are generated from `.td` by the LLVM *tablegen* tool. All C++ files are then compiled and linked together with other LLVM libraries and the LLVM backend can be used for compilation for the architecture originally defined as a CodAL model.

3.2.3 Extensions to the LLVM Framework

The LLVM compiler and the Backend generator were also extended with several features and optimizations by the students of the Faculty of Information Technology at Brno University of Technology as part of their bachelor and master theses. The theses were supervised the author, so they are listed here. The most important extensions that were then included in the compilation infrastructure and the Backend generator are the following:

- VLIW support, profile-based superblock forming and scheduling [78]: This work provided the Backend generator with automatic VLIW support and instruction bundling; the student also implemented advanced scheduling optimizations both for scalar and for VLIW architectures. Instruction scheduling for VLIWs takes instruction latency into account, for example compared to the Tenslicia Xtensa compiler that ignores instruction latencies for VLIWs. Hazards in Xtensa are then resolved only with pipeline stalls.

- Andersen algorithm-based whole program alias analysis and improvement of VLIW scheduling [9]: The biggest performance limitation when compiling for VLIW architectures was the very weak alias analysis present in LLVM; the student implemented an advanced alias analysis algorithm that allows much better scheduling by removing unneeded dependencies. Also in the course of this work, the scheduling algorithms implemented previously in [78] were extended and improved.

- SIMD support in the LLVM compiler [115]: The student extended the existing implementation of autovectorization algorithms in LLVM by adding pragmas, so that the programmer can guide the autovectorizer for better optimizations. He also prepared models and numerous tests for SIMD support, and also helped with SIMD support in the Backend generator and Semantics extractor.

- Peephole optimizer [76]: The LLVM instruction selector cannot automatically use instructions with multiple results. The student implemented an automatic peephole pass generator. This pass can after instruction selection replace simpler instructions with instructions that produce multiple results.

3.2.4 Design Principles of Backend Generator

Only Stopping On Fatal Errors

Similar to the Semantics extractor, the Backend generator should stop only on fatal errors. This way the user can get to the first working version quickly and then start working on optimizations, and explore why certain instructions are not used by the compiler, etc.

Completeness on the Target Instruction Set

One issue that was also explored is the completeness of the instruction set with regard to the operations of the C language. In fact all the nodes that can appear in the LLVM Selection DAG must be covered by an instruction of the instruction set. In a master's thesis [83] this problem was approached by checking the inclusion of 2 tree grammars. The first tree grammar contains all instruction selection patterns, and the second grammar describes tree patterns that can appear in the Selection DAG. However, this has proved to be a too complex problem, and the result was not used in the end. Instead of complex completeness checking, we used several manually implemented checks and the instruction equality rule library to list all the necessary operations. The remaining missing instructions are found by compiling programs from large testsuites. This approach has proved to be fully sufficient.

Automatic Testing and Stability

A high-level language compiler is a very complex piece of software where a change in one part can cause problems in an unrelated part. Especially in a retargetable compiler, fixing one problem for a particular architecture can cause other architectures to fail. To be able to cope with this kind of problems, nightly testing is essential, so that new problems are known as soon as possible. This, however, imposes high requirements on the testing infrastructure, so some basic testsuites should be rather smaller, and full testsuites need not be run every day. It is important to test as many architectures as possible.

Another problem we often encoutered is the Backend generator stability. The diversity of the Backend generator inputs (the instruction semantics) is very high and carefully crafted analyses may fail for some unexpected inputs. The Backend generator must also print an understandable message when the input is incorrect (e.g. an essential instruction is missing). To be able to assure that at least the already fixed problems will not reappear, the author used the classical approach for regression testing. The Backend generator is tested with a set of currently 80 CodAL models and the expected output is compared with the actual output. Stability is also improved by using as many different models as possible, so the author created new instruction accurate CodAL models and initiated the creation of other new models with diverse, previously untested architectural features.

3.2.5 Generated Compiler Results

The Backend generator currently generates architecture files for LLVM version 3.4. The previously used versions of LLVM were 2.8, 3.0, and 3.2. The Backend generator can fully automatically generate a backend for architectures listed in Table 3.2. The Backend generator is also automatically tested on these architectures on a set of tests in the C language from the GCC torture testsuite, LLVM Testsuite, and a set of in-house developed tests. Testing is run over a range of diverse Linux distributions, and also on Microsoft Windows. Each of the architectures has a listed number of tests that are run for it, and also an average of failing tests for running the tests with optimization levels -O0, -O1, -O2, and -O3. A test is first compiled, then assembled and linked with a pre-compiled Standard C library (if available), and then simulated on a simulator generated from the same instruction accurate CodAL model that was used to generate the backend.

For architectures where a standard C library Newlib is available, the number of tests that are run is 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 in fact compile any complex application correctly. The Table 3.2 contains values from testing obtained on 20th of August 2014.

For architectures, where a cycle accurate CodAL model is available, the tests are also run automatically on a cycle accurate simulator. Results for instruction and cycle accurate simulators are generally the same. 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.

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

We will now compare the generated code performance on the MIPS version Release 1 architecture. For this comparison, the following compilers were used: the generated LLVM 3.4-based compiler from a MIPS CodAL model (only GEN LLVM in the following), then the hand-written compiler for MIPS present in LLVM 3.4 (MIPS LLVM), and MIPS GCC 4.9 (MIPS GCC). The source codes of applications come from the LLVM testsuite, which is a collection of tests and many diverse benchmarks. The procedure to obtain these tests was as follows: compile the benchmark sources into an assembly code with all of these three compilers, then assemble with the generated assembler, and link with the standard C library. The standard C library Newlib was compiled with the generated LLVM compiler, but library calls are almost not used, so it has only a minimal impact on the resulting performance. Codasip libraries and tools were used in order to execute the resulting binary on the generated interpreting simulator. Codasip Framework version 2.1.77 was used to generate the assembler and the simulator.

MIPS always executes one instruction after a jump or call instruction. The place for this instruction is called the jump delay slot. A simple optimization to fill jump delay slots for the MIPS architecture was added manually to the generated backend. Both GCC and LLVM MIPS backends contain this optimization too.

The generated assembler and the linker do not support addressing the global variables through a global pointer, so the option `-G 0` was used for MIPS GCC. For MIPS LLVM the options `-mno-check-zero-division` (no checking of zero division, because neither GCC, nor the generated compiler does this), and `-relocation-model=static` were used (MIPS LLVM generated indirect calls to functions from other modules). Not using a global pointer register to access global data has a minimal impact on performance, because instructions to load a global symbol address are always optimized away from loops.

All applications were compiled with the optimization level -O3. Some applications could not be correctly compiled by GCC and linked with this procedure, so these applications have no data for GCC in the graphs.

The relative performance is shown in Figure 3.9. Taken as the baseline was the count of simulated clock cycles for GEN LLVM. The percentage values for MIPS LLVM and MIPS GCC were obtained by dividing the number of clock cycles for the generated compiler by the clock cycles for MIPS LLVM or for MIPS GCC. Higher percentage means higher relative performance, i.e. the higher, the better.

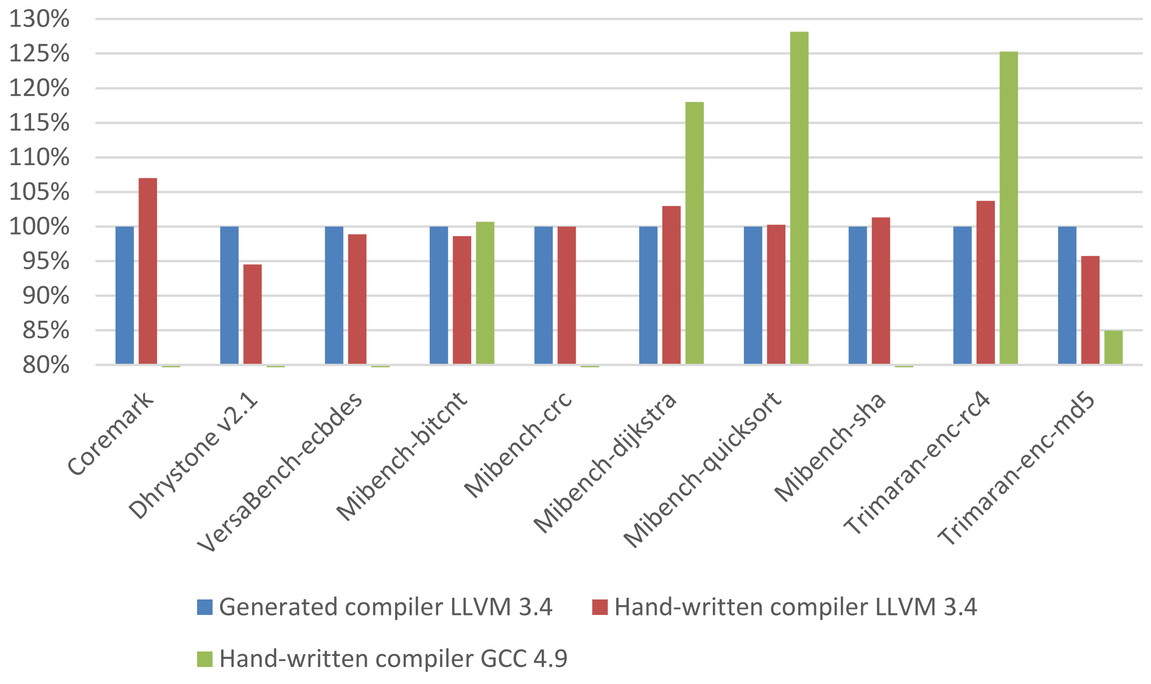


Figure 3.9: Comparison of compiler performance for the MIPS architecture (higher is better)

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

The corresponding relative code size for these benchmarks is shown in Figure 3.10. The code sizes for Mibench-bitcnt and Mibench-crc are very small (approx. 200 bytes), so the relative difference is high. A lower percentage means a lower code size, i.e. the higher, the worse.

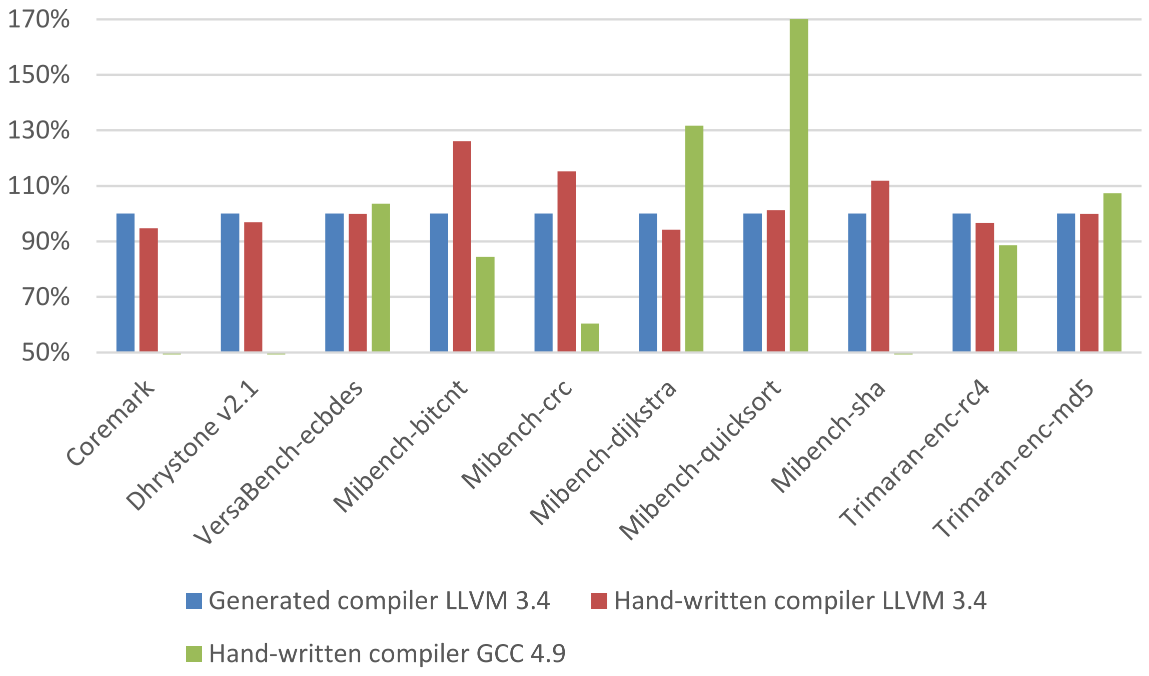


Figure 3.10: Comparison of code size for the MIPS architecture (lower is better)

Figure 3.11 shows a comparison of the performance of the generated compiler from an Open RISC CodAL model with that of a hand-written compiler based on GCC. The graph shows the relative performance calculated from the cycles needed to execute the benchmark, where the GCC compiler was taken as the baseline. A higher percentage meansa higher relative performance.

Results for another processor core Codix RISC are also presented in the following section 3.3. Appendix A lists all the features that are automatically supported by the generated C/C++ compiler.

The Semantics extractor together with the Backend generator performs many tasks fully automatically, and can generate a C compiler backend for many architectures. To create a compiler backend by modifying the backend sources can take more than one month even to someone who knows the retargetable compiler infrastructure. On the other hand, to create a CodAL model and generate a backend using the developed Semantics extractor and Backend generator takes only 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 the LLVM internals. Only when some specific optimizations are needed, can the user add some extension manually.

Compared to the only published and commercially used solution that uses the LISA

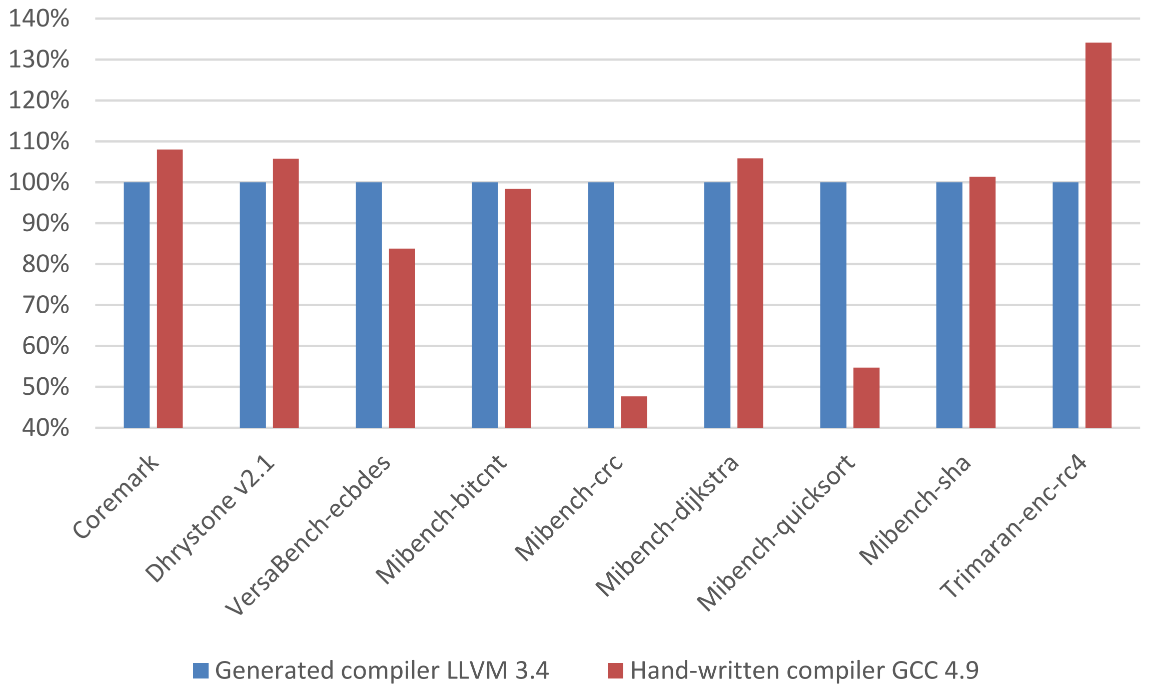


Figure 3.11: Comparison of compiler performance for the Open RISC architecture (higher is better)

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 of the compiler being generated automatically is the ease of adding and removing instructions. If an instruction selection pattern can be generated for it, a new instruction can be used fully automatically by the C compiler. A very fast design space exploration and optimization of the instruction set can also be made. For example, the user can decide to optimize on area by removing the instructions, which is followed by merely regenerating the compiler and running the benchmarks.

Despite the automatic compiler generation, the generated compiler provides a performance comparable with hand-written solutions. Several extensions, in particular for VLIW architectures and SIMD instructions, were also implemented. These extensions can provide even a higher performance than the existing compilers do.

3.3 Reconfigurable Processor Cores

Now, when we have the C compiler generator, the next component of the ASIP design tool is a processor template. This chapter describes processor cores designed by the author and modeled in the Codasip Framework as instruction accurate CodAL models. The author also designed a high-level microarchitecture of all the processor cores while a detailed implementation of cycle accurate CodAL models for Codix STREAM and Codix RISC was made in close cooperation with the author by Marián Pristach from the Faculty of Electrical Engineering and Communication, Brno University of Technology, and that for Codix uRISC was made by Hynek Blaha from the Faculty of Information Technology, Brno University of Technology.

This chapter describes each of these cores, with the explanation of their instruction set and microarchitecture design. Comparisons with other processor cores are also shown.

3.3.1 Minimalistic Architecture Codix uRISC

Codix uRISC [43] was primarily designed as a tutorial model that has just enough instructions to generate a fully working LLVM-based C compiler.

Its instruction set is loosely based on MISP, where several instructions have been removed. Appendix B gives the instruction set of Codix uRISC, which is an example of a minimal instruction set definition needed by any C language program for the LLVM compiler backend.

3.3.2 Streaming Processor Codix STREAM

Codix STREAM [40] is a 16-bit processor primarily designed for dataflow-oriented programming models.

Instruction Set

The ISA (Instruction Set Archtiecture) provides 16 16-bit general-purpose registers, and 15 special registers. Special registers are used for the 32-bit accumulator, zero-overhead loops, autoincrement, and modulo addressing modes, and for higher results of a 32-bit multiplication.

In many processor architecture books such as [34], emphasis is put onto instruction set orthogonality. ISA orthogonality means that any addressing mode should be usable by any instruction, if possible.

Codix STREAM provides 11 addressing modes for instruction operands: 1) constant (a 5-bit signed constant), 2) register, 3) special register, 4) FIFO input, 5) FIFO output, 6) indirect load (load from an address specified by the special register `src_ptr`), 7) indirect store (store to an address specified by the special register `dst_ptr`), 8) autoincrement load (with optional modulo addressing), 9) autoincrement store (with optional modulo addressing), 10) indirect register load (load from an address specified by a general-purpose register), and 11) indirect register store (store to an address specified by a general purpose register). Additional addressing modes can be added.

Most instructions can use any of these addressing modes and any of three operands (destination, source 1, source 2), with some combinations being forbidden. Most arithmetic operations then have a modifier that specifies whether the result should be added to the accumulator.

Lessons Learned

When designing this instruction set, the plan was that the user should write an assembly code for time-critical parts of an application. This way, special features such as accumulator, zero-overhead loops, and addressing registers can be used. However, in practice these features were never used.

The peephole optimizer present in the current generated backend could in some cases use, for example, the accumulator, multiplication with 32-bit result, and addressing with autoincrement. At the time when Codix STREAM was designed, only less than a half of the instructions provided could be used by the compiler. To use the zero-overhead loops automatically, a special optimization written manually in LLVM would be needed.

The main lesson learned from the design and use of Codix STREAM was that it makes no sense to design an extensible processor with features that the compiler cannot use automatically. The user can add special instructions, addressing modes, and registers when needed, because it is hard to predict which features in an ASIP could be generally useful. These extensions can then be used through inline assembly, if they cannot be used by the compiler.

Codix STREAM also has a Harvard architecture (separate addressing spaces for data and code). Having separate addressing modes complicates the programming model of the processor, and also for an FPGA implementation it brings no advantage in Codix STREAM, because the block RAMs present on FPGAs are dual-ported. Thus an instruction can be fetched, and also the memory can be accessed with load or store instructions in one cycle. If separate memory modules are needed for the code and data, it is better to place both of them in one address space, but in different address ranges.

Results

One example of a successful application of the Codix STREAM is a dataflow platform that was used on exhibitions as a demo of sobel filter that detects edges in an image. The platform shown in Figure 3.12 contains 6 Codix STREAM processors connected with FIFOs. The data beiong processed slow through these FIFOs. When a FIFO is full, the processor that wants to push to it is stalled. Similarly, reading from an empty FIFO stalls the reading processor. This way the processors do not need any additional synchronization.

This platform with unmodified Codix STREAM cores, where each core runs on 100MHz (on Virtex 5 FPGA), can process 18 frames per second for a 640x480 pixels input video stream. This sobel filter application can be accelerated with one special instruction to run 5.2 times faster.

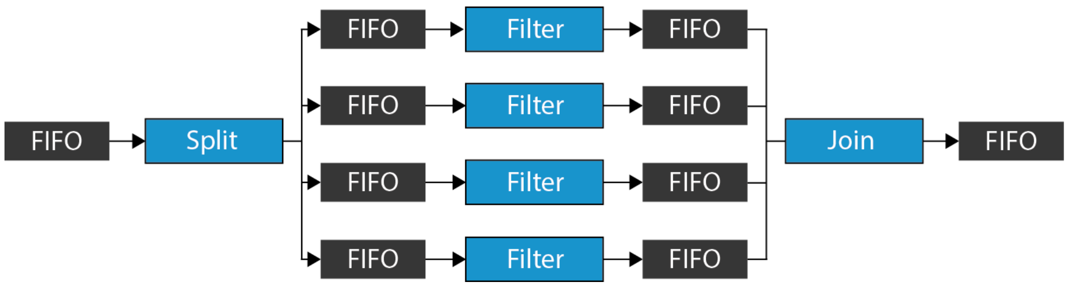


Figure 3.12: Dataflow multicore platform with Codix STREAM

Codix STREAM was designed as a small 16-bit processor with limited addressable memory. The C language standard defines that the base integer type for 16-bit architectures has 16 bits. Existing applications are mostly written in the C code, which requires 32-bit integers, so porting applications to a 16-bit core would be too much work. This was one of the reasons why the author started to work on another, this time 32-bit processor core.

3.3.3 Extensible Processor Core Codix RISC

Codix RISC [41] is a 32-bit RISC architecture whose instruction set is based mainly on the internal LLVM representation [68] and was designed to be highly extensible. New instruction set extensions, special registers, and local memories can be easily added. The current implementation of Codix RISC uses a 7-stage pipeline; its high-level structure is shown in Figure 3.13. A simple jump predictor that assumes that all jumps won't be taken is used.

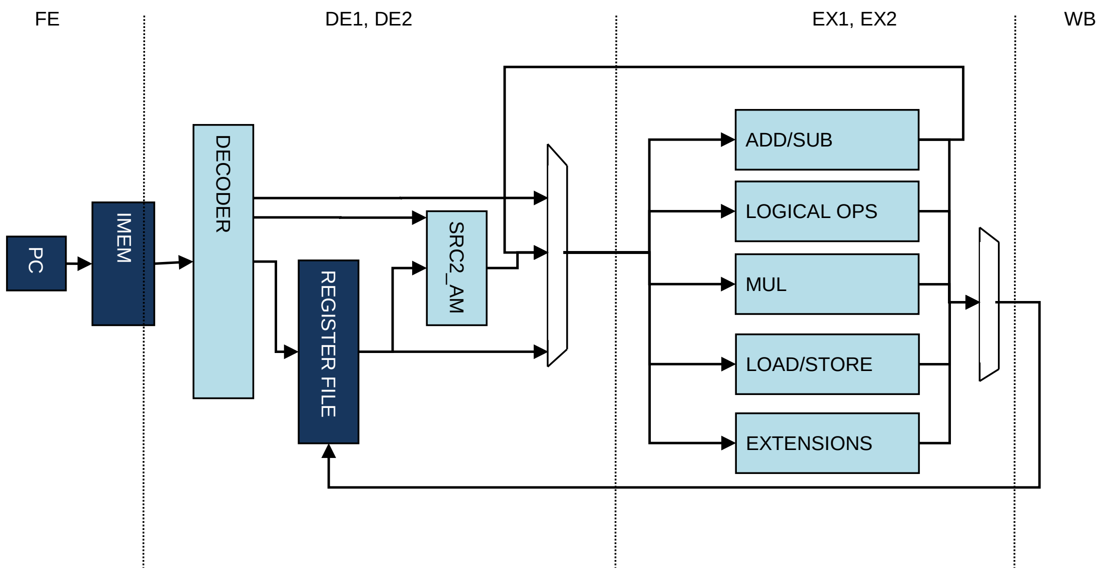


Figure 3.13: Microarchitecture of the extensible processor Codix RISC

One of the biggest differences to other architectures is its interlock-free implementation, where most of the data hazards are handled by the automatically generated C compiler. This facilitates adding new features while simplifying the microarchitecture.

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. Little endianness is useful when porting existing applications, because most applications were primarily written for little endian systems (x86, and ARM), and it is not always that these applications are portable with regard to memory ordering.

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

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

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

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

Interrupts are also supported in Codix RISC. One bit in the status register specifies whether interrupts are enabled, and instructions are provided to call an interrupt and to return from interrupt. Support for exceptions is planned, but it has not been implemented yet.

Structural Hazards

In its first version, Codix RISC did not handle structural hazards. This task was left up to the compiler scheduler, which had to assure that no two instructions would write into the register file simultaneously. This has proved to be an unrealistic expectation, because with stalls from cache and with interrupts, the original instruction schedule from the compiler does not hold. The original idea behind handling structural hazards in a compiler was the expectation that during a stall, all pipeline stages are stalled and no other instruction can appear in the pipeline.

However, this is not how, for example, data cache misses are handled. The processor pipeline generates *stalls* [34] all the stages before the stage using a data cache. The subsequent stages are still running and are finishing the instructions that were already in the pipeline. Instruction ordering at the time the register file is being written is different from what the compiler scheduled. Due to this, hazards on the only write port of the register file have occurred.

Also with interrupts, you can have a slow instruction in the pipeline, in which case the code that is being executed changes, and the original schedule again does not hold. Structural hazard handling was therefore added to the microarchitecture. It has proved to be very cheap from the area perspective.

Data Hazards

The second design decision to handle data hazards has turned out to be very useful. From the assembly code, the user can immediately see whether there were any scheduling problems. For example, that a slow load instruction is followed by NOPs, and that the compiler was not able to make a good schedule. The programmer can then rearrange their code to obtain higher IPC (instructions per cycle).

For all other architectures that handle data hazards, this is hidden and not clearly visible in the code.

This has also led us to explore in more detail the problematics of instruction scheduling and issues that prevent the compiler from finding a good schedule such as a very weak alias analysis in the LLVM framework.

However, using only 32-bit instructions and including NOPs in the code to avoid data hazards make the code size larger. For systems with external flash memories, this is not an issue since flash chips are very cheap and the slight difference (e.g. 0.5 MB) in size is negligible. But when the flash memory is a part of the chip, every kilobyte may count and this may be a problem for a potential user. To deal with the code size issue, a revision of the Codix RISC architecture is planned that will include also 16-bit instructions.

Area and Performance

The Codix RISC synthesis results are shown in Table 3.3. The synthesis was carried out using the Xilinx ISE WebPack 14.2, with the VHDL code generated from a Codix cycle-accurate model with Codasip Framework 1.8.1 [21]. The family is a particular Xilinx FPGA type, Speed specifies the speed grade selected for the synthesis, LUTs are the look-up tables used, Flip Flops specify the 1-bit memories used, and *fmax* is the maximal frequency recommended for the design by the 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 Codix RISC extensible processor core

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

In Table 3.4 Codix RISC is compared with other soft-processors using the Coremark benchmark. Coremark is an application measuring the pipeline throughput [26]. The benchmark consists of three kernels, the first one is a finite state automaton, the second is a pointer-chasing code that manipulates with linked lists, and the third one performs vector and matrix computations. The Coremark benchmark was compiled with an automatically generated C compiler based on LLVM 3.0 with optimization -O3 and simulated on a cycle accurate simulator (with simulated 8kB 4-way caches for instructions and for data) using the Codasip Framework 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 [3].

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

The Codix RISC architecture was also compared with the Microblaze, ARM, 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 the Codasip intersim 3.0.1 was used while for other cores simulators from Open Virtual Platforms build 20130630 were used [48]. 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 the compiler disables the generation of additional NOPs, mainly for load instructions. All benchmarks were compiled with the optimization level -O3. The 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. A comparison is shown in Figures 3.14, 3.15, and 3.16. The WPO values are the results when using the whole program optimization in LLVM. With WPO the whole application is linked together at the LLVM IR level, and additional intraprocedural optimizations (IPO) such as inlining are applied. The result for ARC with Dhrystone 2.1 is missing, because the compiled application could not be correctly executed. The result for Microblaze with Coremark was omitted, because it did not match the results from Table 3.4 (it was on 40% of the performance of other cores), assuming that there was a mistake in the measurement. On the other hand, no mistake in the measurement of Dhrystone 1 on Microblaze was found, so this result is shown.

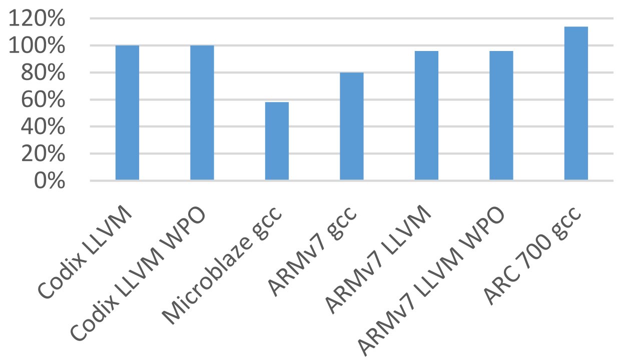


Figure 3.14: Comparison for Dhrystone 1 on simulators (higher is better)

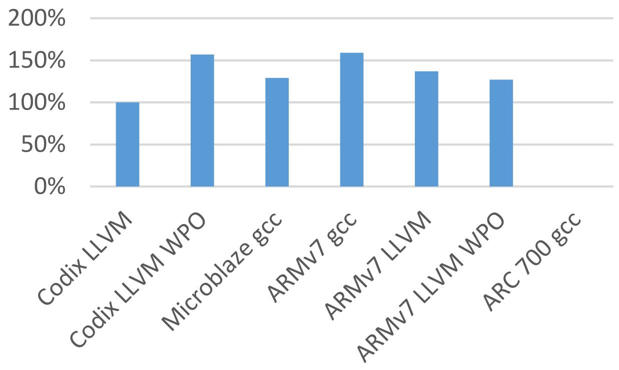


Figure 3.15: Comparison for Dhrystone 2.1 on simulators (higher is better)

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

Instruction Set Optimizations

Two years after the initial design, some experiments with optimizing the base instruction set were made and here are some of the results reported. The goal was to remove some

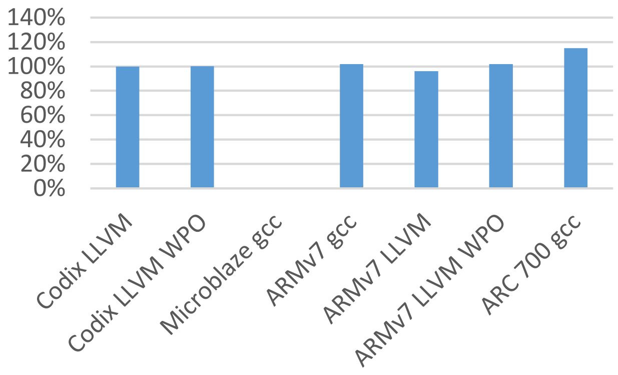


Figure 3.16: Comparison for Coremark on simulators (higher is better)

unused instructions to reduce the area or to modify the existing instructions to improve performance.

Experiments were run on 70 selected benchmarks from the LLVM testsuite and the average and the biggest improvements or penalties are reported. Cycle counts were obtained from an instruction accurate model, where the compiler was instructed to handle data and structural hazards, and the jump penalty was also counted. This way, we were able to obtain precise cycle counts even with an instruction accurate simulator.

The most important changes to the architecture were these:

- Faster jumps: Forwarding the program counter value in order to reduce the jump penalty by one cycle would create a new critical path in the design. This attempt was to simplify allowed comparisons, where 2 registers could be arbitrarily compared with only comparisons to zero. Such comparison does not need an adder. In the Codix RISC model the penalty of a taken jump was also reduced by 1. It was expected that this can improve the performance, but the effect was rather negative.

- Faster addition: Addition is a very common operation and the current addition in Codix RISC has a latency of 2 cycles. By having a faster adder, the result could be forwarded and used directly by the next instruction. This has proved to be a very beneficial optimization and will be implemented in the hardware design in a new revision.

- No indexed stores: This was a first step in removing the third reading port from the general purpose register file. Codix RISC provides indexed store instructions that do this operation: `mem[reg + reg] = reg`. This instruction was removed. The effect was rather negative, although the compiler could generate a better code without this instruction in some cases. The cause of generating better code has not been explored yet.

- Only 2 register read ports: The results are almost the same as for the case without indexed stores. This means that other instructions that use the third register reading port could be easily removed and this is planned for the next Codix RISC revision.

- Load/store with unsigned offset: Normally, to load a global variable is a combination of the instructions lui, ori, and load needed. The lui and ori instructions prepare a 32-bit address in a register that is the used by the load. An optimization to use just the lui and load operations is possible. It would mean adding new variants of load and store instructions that use also 16-bit unsigned offset in addition to the existing loads and stores, which used 16-bit signed offset. However, the average improvement was rather negligible, because the lui and ori instructions are mostly moved out of the loops by loop invariant code motion optimization.

Change in the base architecture	**Average improvement (%)**	**Highest speedup (%)**	**Lowest speedup (%)**
Faster jumps	100.0	104.3	88.3
Faster addition	105.4	113.6	100.2
No indexed stores	99.3	104.1	79.2
Only 2 register read ports	99.7	104.1	79.2
Load/store with unsigned offset	100.2	103.1	100.0

Table 3.5: Optimization of the base instruction set of Codix RISC using generated C compiler

The improvements for cases when some instructions were removed are caused by imperfect LLVM instruction selector that cannot find an optimal coverage and sometimes uses instructions with higher cost (latency) when a combination of instructions with lower cost is available.

From these results we planned to make the addition instruction faster and to remove all instructions that use 3 register inputs, excepting the indexed store.

This subsection also presented an example how automatic C compiler generation can help with instruction set design and optimization. To manually modify the C compiler, assembler and simulator would take much more time and only limited opportunities for optimization could be explored.

Codix RISC - Conclusion

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

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

3.3.4 Extensible VLIW Cores

The author has also designed another two VLIW processor architectures and they are briefly described here.

VIX

The first processor, called VIX [42], was designed as a base for a new VLIW processor. It was first created as a single-issue processor using an instruction accurate model, and after optimization, it was to be changed to a VLIW architecture. The plan was to include initially as many instructions and addressing modes in the architecture as possible, and then, using a large set of benchmarks, remove the instructions and combinations that were used rarely. Some constructions were also added that the author thought would be useful, but were not yet supported by the compiler generator. The architecture uses 24-bit and 40-bit instructions. The long format allows using 64 general purpose registers, and long immediate operands. The short format is then useful for smaller code sizes. The architecture also provides 8 1-bit predicate registers, because VLIW architectures can use predication (conditionally execute an instruction) [27] to execute multiple control flow paths simultaneously.

Support for the new features such as complex addressing modes (for example, by masking the address), and predicate support are still being implemented in the compiler generator.

The VLIW version was planned to have 4 slots (issue width). Also *bundling* was planned; bundling compresses bundles by removing NOPs. Each bundle can have from 24 to 160 bits. Without bundle alignment, each bundle can be placed at any almost arbitrary address, so the instruction memory with the decode stage will have to allow loading 20 unaligned bytes (160 bits), which will be very costly in terms of area.

In the end, we decided to keep the VIX architecture only as a testing model, and to design a simpler VLIW processor.

Codix VLIW

The Codix VLIW [44] architecture was partially inspired by the commercially successful Hexagon processor from Qualcomm [93]. The architecture has 32 32-bit general-purpose registers. There are also 8 1-bit predicate registers. An instruction can be predicated only with predicate registers 0-3, because there is no space in the binary format to use all 8 registers. The remaining registers can be used as temporary registers with other instructions, e.g. to compute a more complex condition for predication.

Due to having only 32 registers, the architecture can have fixed-size 32-bit instructions with 3 register operands. Instruction bundling is also employed, but due to the fixed instruction size, bundles in the memory are always aligned to 32 bits. The Codix VLIW architecture is much simpler than the VIX architecture. The only complex addressing modes that remained are used for loads that often form a critical path for the compiler scheduler, and must be executed as quickly as possible. This architecture also contains only instructions that can be used directly with the current compiler generator (with the exception of predicated execution support).

At the time of finishing this thesis, this architecture is being implemented as an instruction accurate CodAL model.

3.4 Manual ISE identification

Two bachelor theses on automatic Instruction Set Extension (ISE) identification were led by the author. One of them used the *single cut* [74], the other used the *ISEGEN* [112]. These theses were quite successful, but the speed-up obtained was rather limited.

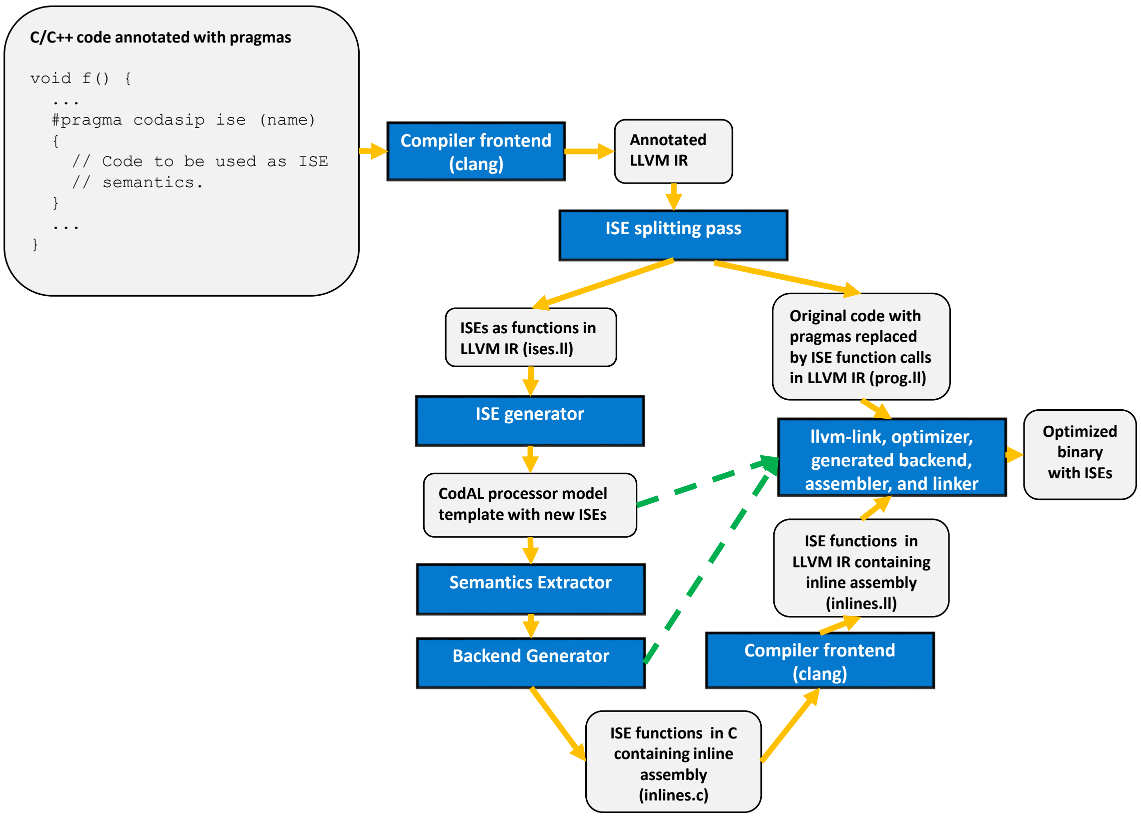

C/C++ code annotated with pragmas
void f() {
...
#pragma codasip ise (name)
{
// Code to be used as ISE
// semantics.
}
...
}
Compiler frontend (clang)
Annotated LLVM IR
ISE splitting pass
ISEs as functions in LLVM IR (ises.ll)
Original code with pragmas replaced by ISE function calls in LLVM IR (prog.ll)
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 (inlines.ll)
Semantics Extractor
Backend Generator
Compiler frontend (clang)
ISE functions in C containing inline assembly (inlines.c)

Figure 3.17: Scheme of support for manually identified ISEs

With regard to other published results on automatic ISE identification (see 2.4.2), the author decided to focus mainly on the automation of user-guided identification. The scheme of this user support is shown in Figure 3.17.

The user first marks the interesting parts of the code with pragma `codasip ise`. The C code is then compiled by a frontend to LLVM IR, pragmas are kept in the LLVM IR as code annotations. Then an ISE splitting pass is run. It removes annotated blocks of code from the application and replaces them with calls to functions with the name prefix `__ise`, the result is stored in the file `prog.ll`. Another LLVM IR file, `ises.ll`, is created, which contains the removed code blocks as functions. This `ises.ll` file is processed by the ISE generator, which generates one instruction for each of the ISE functions in the CodAL model template.

This model is then used to generate compilation tools, and the Backend generator also generates the 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 an inline assembler is necessary.

The file `inlines.c` is then compiled with a frontend into the LLVM IR file `inlines.ll`. It is then linked together with `prog.ll` and optimized. The optimization 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. The 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 were implemented by the author. Used as a processor template was the Codix RISC model. The ISE splitting pass was implemented in a bachelor thesis [75] supervised by the author. The student also made the whole infrastructure work and added other necessary transformations and extensions such as look-up table support. There are currently not many results on using this infrastructure, because it was finished only recently, but from some quick tests, speed-ups from 1.5 to 7.8 for different benchmarks can be obtained very quickly.

Chapter 4

Conclusion

As given in the introduction, the goal planned was to create a complete environment for the optimization of processor cores via using processor templates.

The largest part of this goal was to create a compiler generator. Using the Semantics Extractor and Backend generator, an 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 a faster code. Also many extensions, mainly for VLIW architectures, were implemented that are better than the existing solutions. The resulting compiler is used commercially in the Codasip spin-off, which started from the Lissom project.

The author designed and implemented the Semantics Extractor with its output format, and led the work 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. the LISA ADL), just one instruction semantics representation is needed. As a result, an easily analyzable model of the instruction set is produced that has proved 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 the quality of both 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 [38] owned by Brno University of Technology, and [89] owned by Codasip. Both patents are currently pending.

Finally, the scheme shown in Figure 3.17 (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] ACE: CoSy compiler development system.
http://www.ace.nl/compiler/cosy.html (August 2014), 2013.

[2] Aho, A. V.; Ganapathi, M.; Tjiang, S. W. K.: Code Generation Using Tree Matching and Dynamic Programming. *ACM Trans. Program. Lang. Syst.*, year 11, nr. 4, October 1989: pp. 491–516, ISSN 0164-0925, doi:10.1145/69558.75700.
URL http://doi.acm.org/10.1145/69558.75700

[3] Andersson, S.-A.: Four soft-core processors for embedded systems.
http://www.eetimes.com/design/microcontroller-mcu/4404578/Four-soft-core-processors-for-embedded-systems (August 2014), 2013.

[4] ARM: ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, Issue C. 2014.

[5] Ashenden, P. J.: *The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon) (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., third volume, 2008, ISBN 0120887851, 9780120887859.

[6] Associated Compiler Experts: SuperTest: More than compiler validation.
http://www.ace.nl/sites/default/files/SuperTest.RR_.pdf (August 2014), 2014.

[7] Atmel: AVR32 Archterture Document. 2011.

[8] Balarin, F.; Chiodo, M.; Giusto, P.; aj. (editors): *Hardware-software Co-design of Embedded Systems: The POLIS Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 1997, ISBN 0-7923-9936-6.

[9] Baručák, R.: *Optimalizace v překladači C pro VLIW architektury*. Master's Thesis, FIT, Brno University of Technology, Brno, 2014.

[10] Biswas, P.; Banerjee, S.; Dutt, N.; aj.: Performance and energy benefits of instruction set extensions in an FPGA soft core. In *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, Jan 2006, ISSN 1063-9667, pp. 6 pp.–, doi:10.1109/VLSID.2006.131.

[11] Biswas, P.; Dutt, N.; Pozzi, L.; aj.: Introduction of Architecturally Visible Storage in Instruction Set Extensions. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, year 26, nr. 3, March 2007: pp. 435–446, ISSN 0278-0070, doi:10.1109/TCAD.2006.890582.

[12] Blume, H.; Hübert, H.; Feldkämper, H. T.; aj.: 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*, 2002, ISBN 0-7695-1712-9, p. 29.

[13] Cadar, C.; Sen, K.: Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, year 56, nr. 2, February 2013: pp. 82–90, ISSN 0001-0782, doi:10.1145/2408776.2408795.
URL http://doi.acm.org/10.1145/2408776.2408795

[14] Chandra, R.; Dagum, L.; Kohr, D.; aj.: *Parallel Programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, ISBN 1-55860-671-8, 9781558606715.

[15] Charvat, L.; Smrcka, 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)*, Institute of Electrical and Electronics Engineers, 2012, ISBN 978-1-4673-4441-8, pp. 6–12.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=10135

[16] Chuang, W.; Calder, B.; Ferrante, J.: Phi-Predication for Light-Weight If-Conversion. In *In Proceedings of the International Symposium on Code Generation and Optimization*, 2003, pp. 179–190.

[17] Clark, N.; Blome, J.; Chu, M.; aj.: An architecture framework for transparent instruction set customization in embedded processors. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, June 2005, ISSN 1063-6897, pp. 272–283, doi:10.1109/ISCA.2005.9.

[18] Clark, N.; Hormati, A.; Mahlke, S.; aj.: Scalable Subgraph Mapping for Acyclic Computation Accelerators. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-543-6, pp. 147–157, doi:10.1145/1176760.1176779.
URL http://doi.acm.org/10.1145/1176760.1176779

[19] Clark, N.; Zhong, H.; Mahlke, S.: Processor acceleration through automated instruction set customization. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, Dec 2003, pp. 129–140, doi:10.1109/MICRO.2003.1253189.

[20] Codasip: CodAL Manual. Technical report, Codasip, Brno, CZ, 2014.

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

[22] Community, A.: Bytecode for the Dalvik VM.
https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html (August 2014), 2014.

[23] Cores, O.: OpenRISC 1200 IP Core Specification (Preliminary Draft).
http://openrisc.net/or1200-spec.html (August 2014), 2014.

[24] Cytron, R.; Ferrante, J.; Rosen, B. K.; aj.: Efficiently computing static single assignment form and the control dependence graph. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, year 13, 1991: pp. 451–490.

[25] Dimond, R. G.; Mencer, O.; Luk, W.: Automating processor customisation: optimised memory access and resource sharing. In *DATE*, editation G. G. E. Gielen, European Design and Automation Association, Leuven, Belgium, 2006, ISBN 3-9810801-0-6, pp. 206–211.
URL http://dblp.uni-trier.de/db/conf/date/date2006p.html#DimondML06

[26] EEMBC: CoreMark an EEMBC Benchmark. http://www.eembc.org/coremark/ (August 2014), 2014.

[27] Fisher, J. A.; Faraboschi, P.; Young, C.: *Embedded computing - a VLIW approach to architecture, compilers, and tools*. Morgan Kaufmann, 2005, ISBN 978-1-55860-766-8, I-XXVI, 1-671 pp.

[28] Galuzzi, C.; Bertels, K.: The Instruction-Set Extension Problem: A Survey. *ACM Trans. Reconfigurable Technol. Syst.*, year 4, nr. 2, May 2011: pp. 18:1–18:28, ISSN 1936-7406, doi:10.1145/1968502.1968509.
URL http://doi.acm.org/10.1145/1968502.1968509

[29] Galuzzi, C.; Panainte, E.; Yankova, Y.; aj.: Automatic selection of application-specific instruction-set extensions. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*, Oct 2006, pp. 160–165, doi:10.1145/1176254.1176293.

[30] GCC: Function Attributes - Using the GNU Compiler Collection (GCC). http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html (August 2014), 2014.

[31] Gonzalez, R. E.: Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, year 20, nr. 2, March 2000: pp. 60–70, ISSN 0272-1732, doi:10.1109/40.848473.
URL http://dx.doi.org/10.1109/40.848473

[32] Gracia, D. S.; Pozzi, L.; Ienne, P.; aj.: Developing a back-end for the ARM v5 architecture within the machine SUIF infrastructure. Technical report, École polytechnique fédérale de Lausanne, Technical Report RR03-08, 2004.

[33] Hans-Peter Nilsson: Porting GCC for Dunces. ftp.axis.com/pub/users/hp/pgccfd/pgccfd-0.5.pdf (August 2014), 2000.

[34] Hennessy, J. L.; Patterson, D. A.: *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006, ISBN 0123704901.

[35] Hoffmann, A.; Meyr, H.; Leupers, R.: *Architecture exploration for embedded processors with LISA*. Kluwer, 2002, ISBN 978-1-4020-7338-0, I-VIII, 1-230 pp.

[36] Hohenauer, M.; Leupers, R.: *C Compilers for ASIPs: Automatic Compiler Generation with LISA*. Springer Publishing Company, Incorporated, firts volume, 2009, ISBN 1441911758, 9781441911759.

[37] Holloway, G.; Smith, M. D.: The Machine-SUIF SUIFvm Library. Technical report, Harvard University, 2002.

[38] Hruška, T.; Přikryl, Z.; Husár, A.: A METHOD AND AN APPARAUS FOR INSTRUCTION SET TRANSLATION USING FINITE STATE AUTOMATA, US patent pending. 2013.

[39] Husár, A.: *Implementace obecného assembleru*. Master's Thesis, FIT, Brno University of Technology, Brno, 2007.

[40] Husár, A.: *Codix STREAM Instruction Set Reference*. Codasip, 2011.

[41] Husár, A.: *Codix RISC Instruction Set Reference*. Codasip, 2012.

[42] Husár, A.: *Návrh procesoru VIX*. Codasip, 2013.

[43] Husár, A.: *Codix uRISC Instruction Set Reference*. Codasip, 2014.

[44] Husár, A.: *Návrh procesoru Codix VLIW*. Codasip, 2014.

[45] Husár, A.; Hruška, T.; Trmač, M.; aj.: Instruction Selection Patterns Extraction from Architecture Specification Language ISAC. In *Proceedings of the 16th Conference Student EEICT 2010 Volume 5*, Faculty of Information Technology BUT, 2010, ISBN 978-80-214-4080-7, pp. 166–170.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9224

[46] Husár, A.; Trmač, M.; Hranáč, J.; aj.: Automatic C Compiler Generation from Architecture Description Language ISAC. In *6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, Masaryk University, 2010, ISBN 978-80-87342-10-7, pp. 84–91.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9403

[47] IBM: PowerPC User Instruction Set Architecture, Book I, Version 2.02. 2005.

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

[49] Infineon: TriCore V1.6 Instruction Set User Manual (Volume 2). 2012.

[50] Instruments, T.: MSP430i2xx Family: User's Guide. 2014.

[51] Intel: Intel 64 and IA-32 Architectures Software Developers Manual . 2014.

[52] ISO/IEC: Working Draft, Standard for Programming Language C++, N3337. 2011.

[53] ISO/IEC: Information technology - Common Language Infrastructure (CLI). Online, September 2012.

[54] Jakob Stoklund Olesen: Greedy Register Allocation in LLVM 3.0.
http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html (August 2014), 2012.

[55] Jim Grosbach and Owen Anderson: LLVM MC in Practice.
http://llvm.org/devmtg/2011-11/Grosbach_Anderson_LLVMMC.pdf (August 2014), 2011.

[56] Koubek, K.: *Procesorové jádro ADOP*. Master's Thesis, FEL, Czech Technical University in Prague, Brno, 2008.

[57] Krzikalla, O.: Performing Source-to-Source Transformations with Clang. `llvm.org/devmtg/2013-04/krzikalla-slides.pdf` (August 2014), 2013.

[58] Křoustek, J.; Pokorný, F.: Reconstruction of Instruction Idioms in a Retargetable Decompiler. In *4th Workshop on Advances in Programming Languages (WAPL'13)*, IEEE Computer Society, 2013, ISBN 978-1-4673-4471-5, pp. 1507–1514. URL `http://www.fit.vutbr.cz/research/view_pub.php?id=10328`

[59] Lattner, C.: *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's Thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002, *See* `http://llvm.cs.uiuc.edu`.

[60] Leupers, R.: *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools*. Norwell, MA, USA: Kluwer Academic Publishers, 2000, ISBN 0792379896.

[61] Leupers, R.; Karuri, K.; Kraemer, S.; aj.: A design flow for configurable embedded processors based on optimized instruction set extension synthesis. *Design, Automation & Test in Europe Conference & Exhibition*, year 1, 2006: p. 128, doi:http://doi.ieeecomputersociety.org/10.1109/DATE.2006.243972.

[62] Leupers, R.; Marwedel, P.: *Retargetable compiler technology for embedded systems: tools and applications*. Norwell, MA, USA: Kluwer Academic Publishers, 2001, ISBN 0-7923-7578-5.

[63] Leupers, R.; Marwedel, P.: *Retargetable Compiler Technology for Embedded Systems: Tools and Applications*. Norwell, MA, USA: Kluwer Academic Publishers, 2001, ISBN 0-7923-7578-5.

[64] Lin, M.; Yu, Z.; Zhang, D.; aj.: Retargeting the Open64 Compiler to PowerPC Processor. In *Embedded Software and Systems Symposia, 2008. ICESS Symposia '08. International Conference on*, July 2008, pp. 152–157, doi:10.1109/ICESS.Symposia.2008.69.

[65] Lindholm, T.; Yellin, F.; Bracha, G.; aj.: The Java Virtual Machine Specification. `http://docs.oracle.com/javase/specs/jvms/se7/html/` (August 2014), 2013.

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

[67] LLVM: Compiler-rt runtime libraries. `http://compiler-rt.llvm.org/` (August 2014), 2014.

[68] LLVM: LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html` (August 2014), 2014.

[69] LLVM: LLVM TableGen Documentation. `http://llvm.org/docs/TableGen/index.html` (August 2014), 2014.

[70] Martin, K.; Wolinski, C.; Kuchcinski, K.; aj.: Generic environment for design and utilization of reconfigurable application-specific processors extensions. In *University booth at DATE09*, year 8, 2009.

[71] Maxfield, C.: Wow! Tensilica licensees have shipped 2 billion IP cores! http://www.eetimes.com/document.asp?doc_id=1262638 (August 2014), 2012.

[72] Mehdi Kamal, M. P., Ali Afzali-Kusha: Timing variation-aware custom instruction extension technique. *Design, Automation & Test in Europe Conference & Exhibition*, year 0, 2011: pp. 1–4, ISSN 1530-1591, doi:http://doi.ieeecomputersociety.org/10.1109/DATE.2011.5763324.

[73] Mehdipour, F.; Noori, H.; Inoue, K.; aj.: High Performance, Low Power Reconfigurable Processor for Embedded Systems. In *ISOCC*, 2007.

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

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

[76] Ministr, M.: *Peephole optimalizátor pro konfigurovatelné architektury procesorou*. Bachelor's Thesis, FIT, Brno University of Technology, Brno, 2014.

[77] Ministr, M.: *Virtuální platformy pro simulaci instrukčních sad*. Master's Thesis, FIT, Brno University of Technology, Brno, 2014.

[78] Mináč, T.: *Kompilátor jazyka C pro VLIW architektury*. Master's Thesis, FIT, Brno University of Technology, Brno, 2013.

[79] Mishra, P.; Dutt, N. (editors): *Processor Description Languages*. Morgan Kaufmann, 2008, ISBN 0-12-374287-0.

[80] Mondal, S.: *Compiler Back End Generation from nML Machine Description*. Master's Thesis, Indian Institute of Technology, Kanpur, 1999.

[81] Muchnick, S. S.: *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, ISBN 1-55860-320-4.

[82] Murakami, K.: High Performance, Low Power Reconfigurable Processor for Embedded Systems. In *ISOCC*, 2007.

[83] Nagy, M.: *Detekce kompletnosti instrukční sady pro generování univerzálního překladače jazyka C*. Master's Thesis, FIT, Brno University of Technology, Brno, 2012.

[84] Open64: Open64 Compiler Website. http://www.open64.net/ (August 2014), 2014.

[85] Patt, Y.: Requirements, bottlenecks, and good fortune: agents for microprocessor evolution. *Proceedings of the IEEE*, year 89, nr. 11, Nov 2001: pp. 1553–1559, ISSN 0018-9219, doi:10.1109/5.964437.

[86] Plessl, C.: Introduction to the LLVM Compiler Framework. University of Paderborn, Germany, 2012.

[87] Pogde, P.: Retargettable Code Generation using Sim-nML Machine Description. Technical report, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, 2000.

[88] Pozzi, L.; Atasu, K.; Ienne, P.: Exact and approximate algorithms for the extension of embedded processor instruction sets. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, year 25, nr. 7, July 2006: pp. 1209–1229, ISSN 0278-0070, doi:10.1109/TCAD.2005.855950.

[89] Přikryl, Z.; Husár, A.; Masařík, K.; aj.: A METHOD AND AN APPARATUS FOR AUTOMATIC PROCESSOR DESIGN AND VERIFICATION, US patent pending. 2013.

[90] Přikryl, Z.; Křoustek, J.; Hruška, T.; aj.: Design and Simulation of High Performance Parallel Architectures Using the ISAC Language. *GSTF International Journal on Computing*, year 1, nr. 2, 2011: pp. 97–106, ISSN 2010-2283.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9519

[91] Přikryl, Z.; Křoustek, J.; Hruška, T.; aj.: Fast Just-In-Time Translated Simulation for ASIP Design. In *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, IEEE Computer Society, 2011, ISBN 978-1-4244-9753-9, pp. 279–282.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9567

[92] QEMU: QEMU: Open Source Processor Emulator. http://www.qemu.org/ (August 2014), 2014.

[93] Qualcomm: Hexagon V5/V55 Programmer's Reference Manual. https://developer.qualcomm.com/mobile-development/maximize-hardware/multimedia-optimization-hexagon-sdk/hexagon-dsp-processor (August 2014), 2013.

[94] Richard M. Stallman and the GCC Developer Community: GNU Compiler Collection Internals, For gcc version 4.10.0 (pre-release). http://gcc.gnu.org/onlinedocs/gccint.pdf (August 2014), 2014.

[95] Rodriguez, C. S.; Fischer, G.; Smolski, S.: *The Linux(R) Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005, ISBN 0131181637.

[96] Rowen, Chris and Hennessy, John , and Christensen, Clayton M. and Leibson, Steve: *Engineering the complex SOC : fast, flexible design with configurable processors*. Prentice Hall Modern Semiconductor Design Series, Upper Saddle River: Prentice Hall, 2004, ISBN 0-13-145537-0.
URL http://opac.inria.fr/record=b1108184

[97] Schneider Beck Fl., A. C.; Carro, L.: *Dynamic Reconfigurable Architectures and Transparent Optimization Techniques: Automatic Acceleration of Software Execution*. Springer Publishing Company, Incorporated, firts volume, 2010, ISBN 9048139120, 9789048139125.

[98] Smith, J.; Nair, R.: *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005, ISBN 1558609105.

[99] Smith, M. D.: Machine SUIF Project website. http://www.eecs.harvard.edu/hube/software/software.html (August 2014), 2014.

[100] Society, I. C.: *IEEE Standard for Floating-Point Arithmetic*. 2008, ISBN 978-0-7381-5753-5.

[101] Sreedhar, V. C.; Ju, R. D.-C.; Gillies, D. M.; aj.: Translating Out of Static Single Assignment Form. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, London, UK, UK: Springer-Verlag, 1999, ISBN 3-540-66459-9, pp. 194–210.
URL http://dl.acm.org/citation.cfm?id=647168.718132

[102] Stone, J. E.; Gohara, D.; Shi, G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, year 12, nr. 3, May 2010: pp. 66–73, ISSN 0740-7475, doi:10.1109/MCSE.2010.69.
URL http://dx.doi.org/10.1109/MCSE.2010.69

[103] Synopsys: DesignWare ARC Processor Cores. http://www.synopsys.com/IP/PROCESSORIP/ARCPROCESSORS/Pages/default.aspx (August 2014), 2014.

[104] Synopsys: DesignWare ARC Processor Portfolio. https://www.synopsys.com/dw/doc.php/ds/cc/arc_processor_solutions.pdf (August 2014), 2014.

[105] Synopsys: DesignWare ARChitect. http://www.synopsys.com/dw/ipdir.php?ds=sw_architect_config (August 2014), 2014.

[106] Synopsys: IP Designer, IP Programmer and MP Designer. http://www.synopsys.com/IP/ProcessorIP/asip/ip-mp-designer/Pages/default.aspx (August 2014), 2014.

[107] Synopsys: Processor Designer. http://www.synopsys.com/systems/blockdesign/processordev/pages/default.aspx (August 2014), 2014.

[108] Technologies, M.: MIPS32 Architecture For Programmers, Volume II: The MIPS32 Instruction Set. 2003.

[109] Thompson, M.: Overcoming the power/performance paradox in processor IP. http://www.techdesignforums.com/practice/technique/power-performance-processor-ip/ (August 2014), 2014.

[110] Trmač, M.; Hruška, T.: Comparing gcc and LLVM Back-end Infrastructure. Technical report, Brno University of Technology, Faculty of Information Technology, 2008.

[111] Vanbroekhoven, P.; Janssens, G.; Bruynooghe, M.; aj.: A practical dynamic single assignment transformation. *ACM Transactions on Design Automation of Electronic Systems*, year 12, nr. 4, September 2007: pp. 1–12, doi:10.1145/1278349.1278353. URL https://lirias.kuleuven.be/handle/123456789/146589

[112] Česka, M.: *Automatické vyhledávání instrukčních rozšíření aplikačních procesorou.* Bachelor's Thesis, FIT, Brno University of Technology, Brno, 2013.

[113] Verma, A. K.; Brisk, P.; Ienne, P.: Fast, Nearly Optimal ISE Identification With I/O Serialization Through Maximal Clique Enumeration. *Ieee Transactions On Computer-Aided Design Of Integrated Circuits And Systems*, year 29, 2010: pp. 341–354, ISSN 0028-0070, doi:10.1109/TCAD.2010.2041849.

[114] Šimková, M.; Přikryl, Z.; Hruška, T.; aj.: Automated Functional Verification of Application Specific Instruction-set Processors. *IFIP Advances in Information and Communication Technology*, year 4, nr. 403, 2013: pp. 128–138, ISSN 1868-4238. URL http://www.fit.vutbr.cz/research/view_pub.php?id=10268

[115] Šnobl, P.: *Podpora SIMD instrukcí v překladači LLVM.* Bachelor's Thesis, FIT, Brno University of Technology, Brno, 2014.

[116] Whitham, J.; Audsley, N. C.: Integrating Custom Instruction Specifications into C Development Processes. In *ARC*, *Lecture Notes in Computer Science*, year 3985, editation K. Bertels; J. M. P. Cardoso; S. Vassiliadis, Springer, 2006, pp. 431–442. URL http://dblp.uni-trier.de/db/conf/arc/arc2006.html#WhithamA06

[117] Wiki, G.: A Brief History of GCC. [Online] https://gcc.gnu.org/wiki/History, 2008.

[118] Wilson, R.; French, R.; Wilson, C.; aj.: An Overview of the SUIF Compiler System. Technical report, Stanford University, 1995.

[119] Xilinx: MicroBlaze Processor Reference Guide, Embedded Development Kit EDK 10.1i. 2008.

[120] Zhang, J.; Zhang, Z.; Zhou, S.; aj.: Bit-level Optimization for High-level Synthesis and FPGA-based Acceleration. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, New York, NY, USA: ACM, 2010, ISBN 978-1-60558-911-4, pp. 59–68.

Appendix A

Features Supported by the Generated C/C++ Compiler

This appendix lists features that the automatically generated C/C++ compiler supports (by August 2014). If the support is either *automatic* or *OK*, then the user does not have to care about these features, because they are supported fully automatically. *Manual LLVM code* means that support can be added either by simple extension of the generated LLVM sources (in most cases) or by simpler modifications of the LLVM code. *Planned* means that automatic support is planned, and support can be currently added manually. *No* means that there is no support in LLVM for such a feature. It can be added, but the modifications will be very complex.

Feature	**Support**
Standard arithmetic instructions	Automatic
Complex instructions with one result	Automatic - if such patterns appear in the C code
Complex instructions with multiple results	Mostly automatic - if such patterns appear in the C code
Conditional jumps; selects; condition code generation	Automatic - if suitable instructions are found; special language to define equivalences
Calls and returns	Calls that store the return address to a register or to stack are both supported
SIMD instructions	Automatic - if the LLVM vectorizer identifies such instructions; pragmas to guide the vectorizer
Floating point instructions - 32 and 64-bit	Automatic for 32 and 64-bit floats; manual for 16 and 128-bit
Non-standard integers	No
Prologue and epilogue generation	Automatic if suitable instructions are found
Instructions for spilling in register allocator	Automatic if suitable instructions are found; otherwise manual
Register pairs	Manual LLVM code
Indexed registers	Manual LLVM code
Global data pointer	Manual LLVM code
Address calculation from general purpose registers	Automatic
Autoincrement/autodecrement	Mostly automatic - if such patterns appear in the C code
Special registers for address calculation	Manual LLVM code
Special addressing modes such as modulo addressing	Manual LLVM code

Table A.1: Instruction selection

Feature	**Support**
Usage of information about instruction latencies and resource usage	Scheduling information for instructions is specified in a special language; automatic
Possibility to handle data and structural hazards in compiler	OK
Jump delay slots filling	Planned

Table A.2: Instruction scheduling

Feature	**Support**
Instruction bundles	OK
Scheduling for VLIWs	OK
Profile guided superblock formation and superblock scheduling	OK
Profile guided if-conversion at the LLVM IR level	OK
Software pipelining	Under development
Predication and full if-convertsion in the backend	Under development
Clustered register files	No

Table A.3: VLIW features

Feature	**Support**
LLVM Compiler-rt	OK
Newlib	OK
uClibc	OK
Apache C++	Mostly OK; some parts under development

Table A.4: Standard runtime, C and C++ libraries

Appendix B

Minimal Instruction Set Needed by LLVM Backend

This section lists instructions and their semantics from the Codix uRISC architecture. Codix uRISC is a minimalistic instruction set sufficient for the generated compiler to compile any C language program.

In the instruction semantics description in the following Tables, the operators have the same meaning as in the C language. There are also several auxiliary operators: sext32 sign-extends the input to a 32-bit value, zext32 zero-extends the input to a 32-bit value, trunc8 truncates the input to 8 bits, trunc16 truncates the input to 16 bits, not is a binary negation, xor is a logical exclusive or, (u)» is logical (unsigned) right shift, and (s)» is arithmetic (signed) right shift.

Arithmetical and logical instructions (in Table B.1) correspond to the elementary operations from LLVM IR. Division is not needed, because it can be replaced with a software implementation. Conditional moves can be theoretically removed, but they are very useful for the implementation of the ternary operator in C, and for SSA *phi* operation transformation.

Traditional architectures have usually all versions of arithmetic operations with immediate operands, but to compile with LLVM it is sufficient to have just one instruction to put a pointer-wide constant into a register. The `ADDI` instruction is needed by the compiler generator in this form to add or subtract a constant to/from the stack pointer. For small immediates the instruction `MOVSI` can be used instead of the `LUI`, `ORI` combination (Table B.2).

Loads and stores must be able to access a particular byte, halfword or byte in the memory. Certain peripherals can react to access to their control registers, so a byte load cannot be emulated with a whole word load. `LOADUB` does the zero-extension, and `LOADSB` does the sign-extension. Theoretically, only one version is needed, because zero- or sign-extension can be done with arithmetic instructions, but the LLVM backend requires both the sign- and zero-extend load versions of these instructions to be present (Table B.3).

Conditional jumps are relative, while unconditional jumps and calls are absolute. Jump to an address in a register is necessary for the jump tables generated from the `switch` C language constructions, while calls to an address in the register are needed to call function pointers (Table B.4).

Codix uRISC has only 2 special instructions: a no-operation, and a halt (Table B.5).

Syntax	**Semantics**	**Comments**
MOV d s0	d:= s0	
NEG d s0	d:= not s0	
ADD d s0 s1	d := s0 + s1	
SUB d s0 s1	d := s0 - s1	
MUL d s0 s1	d := s0 * s1	The result is truncated to 32 bits.
AND d s0 s1	d:= s0 & s1	
OR d s0 s1	d:= s0 | s1	
XOR d s0 s1	d:= s0 xor s1	
SLL d s0 s1	d:= s0 « s1	Shift left logical
SRL d s0 s1	d:= s0 (u)» s1	Shift right logical
SRA d s0 s1	d:= s0 (s)» s1	Shift right arithmetic
MOVZ d s0 s1	if (s1 == 0) d:= s0	Conditional move
MOVNZ d s0 s1	if (s1 != 0) d:= s0	Conditional move
SETEQ d s0 s1	d:= (s0 == s1)	Set 0 when s0 and s1 are not equal set to non-zero value when s0 and s 1 are equal.
SETNEQ d s0 s1	d:= (s0 != s1)	
SETSLT d s0 s1	d:= (s0 (s)< s1)	
SETULT d s0 s1	d:= (s0 (u)< s1)	
SETSLE d s0 s1	d:= (s0 (s)<= s1)	
SETULE d s0 s1	d:= (s0 (u)<= s1)	

Table B.1: Arithmetical, logical, and compare instructions with register operands

Syntax	**Semantics**	**Comments**
LUI d imm16	d:= imm16 « 16	
ORI d s0 imm16	d:= s0 | zext32(imm16)	Combination of LUI and ORI is used to load a 32-bit immediate.
MOVSI d imm16	d := sext32(imm16)	
ADDI d s0 imm16	d := s0 + sext32(imm16)	

Table B.2: Instructions with immediate operands

Syntax	**Semantics**
LOAD d b + imm16	d := mem[b + sext32(imm16)]
STORE s b + imm16	mem[b + sext32(imm16)] := s
LOADUB d b + imm16	d:= zext32(mem [b + sext32(imm16)].subblock(0 1))
LOADSB d b + imm16	d:= sext32(mem [b + sext32(imm16)].subblock(0 1))
LOADUH d b + imm16	d:= zext32(mem [b + sext32(imm16)].subblock(0 2))
LOADSH d b + imm16	d:= sext32(mem [b + sext32(imm16)].subblock(0 2))
STOREB s b + imm16	mem[b + sext32(imm16)] .subblock(0 1) := trunc8(s)
STOREH s b + imm16	mem[b + sext32(imm16)] .subblock(0 2) := trunc16(s)

Table B.3: Load and store instructions

Syntax	**Semantics**	**Comments**
JUMP imm26	pc:= imm26	Absolute jump
CALL imm26	regs[31] = pc + 4; pc:= imm26	Return address (address of the next instruction) is stored in register r31.
JUMP s	pc:= s	Absolute jump to an address in register
CALL s	regs[31] = pc + 4; pc:= s	Return address (address of the next instruction) is stored in register r31.
JUMPZ s rel_addr16	if (s == 0) pc := pc + sext32(rel_addr16) + 4	Relative jump
JUMPNZ s rel_addr16	if (s != 0) pc := pc + sext32(rel_addr16) + 4	Relative jump

Table B.4: Jump and call instructions

Syntax	**Semantics**	**Comments**
NOP		No operation
HALT	Halt processor	Stops processor or simulation execution.

Table B.5: Special instructions

Appendix C

ASIP Design Methodology

An ASIP design methodology usable with the Codasip Framework is described in this appendix. To use such a methodology in the context of the Lissom project was originally proposed by the author of this thesis and was then expanded and tools were implemented to support this kind of processor development.

The methodology consists of three main steps:

1. design and optimization of the Instruction Accurate (IA) model,
2. design and optimization of the Cycle Accurate (CA) model, and
3. verification of the Instruction Accurate and Cycle Accurate equivalency.

If needed, it is possible to return from step 2 back to step 1 to alter the instruction accurate model.

Design and optimization of the Instruction Accurate (IA) model This step is shown in Figure C.1. The user starts either with a processor template in the form of an IA model or with a definition of the processor architecture. Once an IA model is available, all tools, including the C/C++ compiler, assembler, and simulator, are generated. The C or C++ application is compiled and profiled, using the generated simulator. Parts of the code where most of the time is spent can be optimized by adding new instructions. When new instructions are added to the IA model, all tools are regenerated. The user must have at least some understanding of how the new instructions will be implemented in hardware, for example, they must be able to specify instruction latencies for the compiler and, also, not to design instructions that would be very costly or very hard to implement in hardware. Tools are then iteratively regenerated and new instructions added and removed until the performance requirements are met. With IA models, it is very easy to make modifications and quickly try out new design alternatives.

Design and optimization of the Cycle Accurate (CA) model Once the design of an IA model is ready, it can be modeled as a CA model. The VHDL or Verilog implementation can be generated from CA models as shown in Figure C.2 and this is then used for optimization mainly on area and power. Some instructions that do not offer a good ratio between the area needed and the performance provided can be removed both from the CA and and the IA models. Once the requirements for performance, area, and power are met, designers can move on to the next step of the methodology.

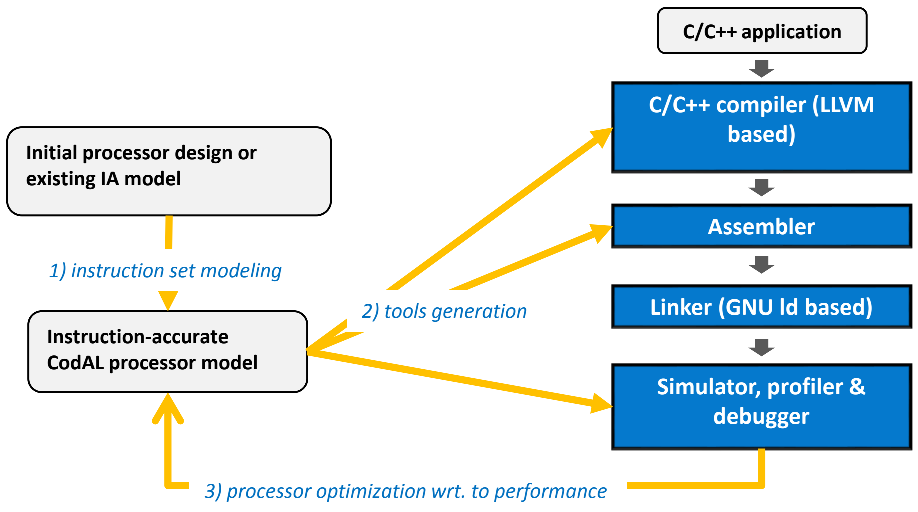

C/C++ application
C/C++ compiler (LLVM based)
Assembler
Linker (GNU ld based)
Simulator, profiler & debugger
Initial processor design or existing IA model
1) instruction set modeling
Instruction-accurate CodAL processor model
2) tools generation
3) processor optimization wrt. to performance

Figure C.1: Step 1 of the ASIP design methodology - IA model design and optimization

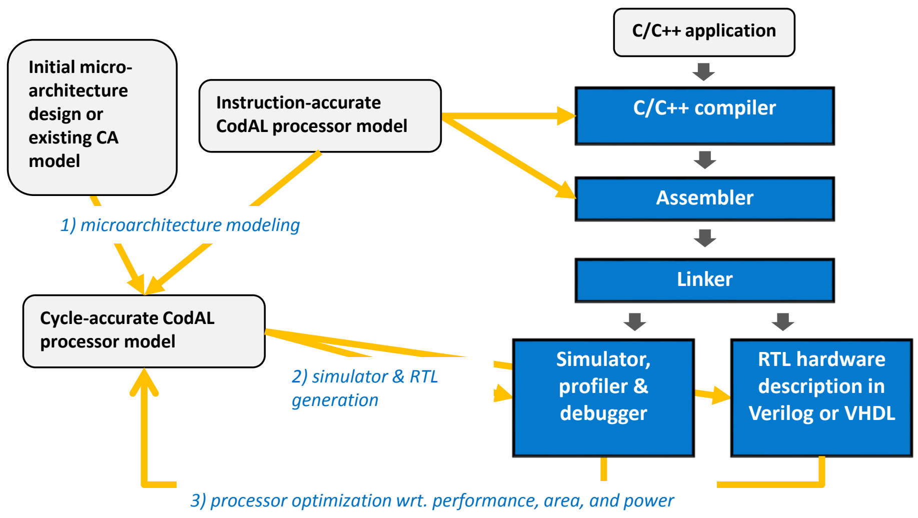

C/C++ application
Initial micro-architecture design or existing CA model
Instruction-accurate CodAL processor model
C/C++ compiler
Assembler
Linker
1) microarchitecture modeling
Cycle-accurate CodAL processor model
2) simulator & RTL generation
Simulator, profiler & debugger
RTL hardware description in Verilog or VHDL
3) processor optimization wrt. performance, area, and power

Figure C.2: Step 2 of the ASIP design methodology - CA model design and optimization

Verification of the Instruction Accurate and Cycle Accurate Model Equivalency
This step in the methodology is to use a set of test cases from standard testsuites, new handwritten tests, and also automatically generated tests to verify model equivalency. The RTL implementation of the optimized processor core generated from the CA model is verified against the compiler and simulator generated from the IA model. This brings also some advantages, because the IA model is very simple and it is hard to make a mistake in it. The CA model can be quite complex and this methodology provides a way how to have a *golden model* as a reference.

The result of this methodology is two verified and optimized processor models that can be used both to generate RTL for hardware synthesis and to generate a whole toolchain to compile, simulate, profile, and debug applications. What is done during processor optimization is in fact minimizing the *semantic gap* between the problem domain and the architecture by introducing new instructions. This then allows more efficient mapping between the problem and the electronic components, as shown in Figure 1.1.

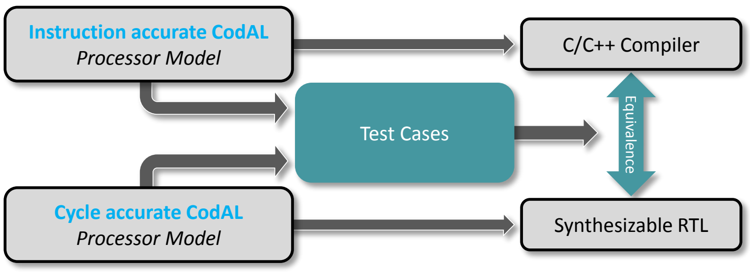


Figure C.3: Step 3 of the ASIP design methodology - verification of IA and CA models equivalence

Appendix D

Optimization of Codix RISC with Instruction Set Extensions

Results for automatic ASIP optimization that were published in the past can, to a certain extent, discourage using an ASIP, because in most cases the results show no more than 2-times higher performance improvements (see 2.4.2). When faced with the choice of using either a standard processor core such as ARM or an ASIP, the chip designer must consider the risk involved in using a non-standard processor core with uncertain hardware implementation and toolchain quality. In these cases, the fact that the ASIP core provides twice as high performance may not justify the risk. Using the ARM core, for example on a higher clock frequency, when the other requirements are still met, is a much safer choice.

This appendix shows the results of optimizing the Codix RISC processor core with instruction set extensions. These optimizations were made using the generated C compiler and the Codix RISC processor core, but without the whole ASIP optimization tool shown in section 3.4, because it was not available at that time. The forms of useful extensions were used to set priorities on what the ASIP optimization tool should support.

The benchmarks that were optimized came from the LLVM testsuite. The instruction accurate CodAL model of Codix RISC was used for the optimization. Figures D.1, and D.2 show the results collected from papers on automatic ISE optimization. Speed-ups from articles are denoted in the following graphs as A1-A13; A1 represents data from article [97], A2 [61], A3 [19], A4 [17], A5 [88], A6 [72], A7 [70], A8 [73], A9 [116], A10 [10], A11 [29], A12 [11], and A13 [113]. The speed-ups for Codasip Framework were obtained by the author using the manual ISE identification method based on step 1 from the ASIP design methodology described in Appendix C.

Figure D.3 shows the dataflow graph of one ISE used in the MiBench rawcaudio benchmark. Such a dataflow graph could be automatically implemented as a pipelined functional unit in the processor hardware.

From the forms of useful extensions it was deduced that ISE identification support must take into account the following features: The first is the usage of special registers that allows higher amount of input and output register operands without the need to increase the general-purpose register file read and write ports. The second is that ISEs must have at least limited access to the main memory, in most cases just one load after address calculation and then one store are needed. The third is wide memory access, e.g. through interleaved memories. Much higher speed-up can be reached when ISEs can load or store 128-bit or wider data in one cycle. The fourth is multiported memories that allow accessing multiple

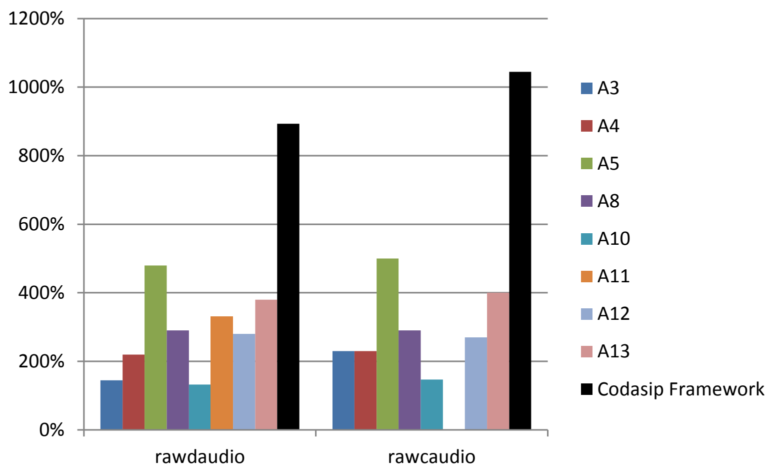


Figure D.1: Comparison of speed-ups for manually and automatically identified ISEs (1)

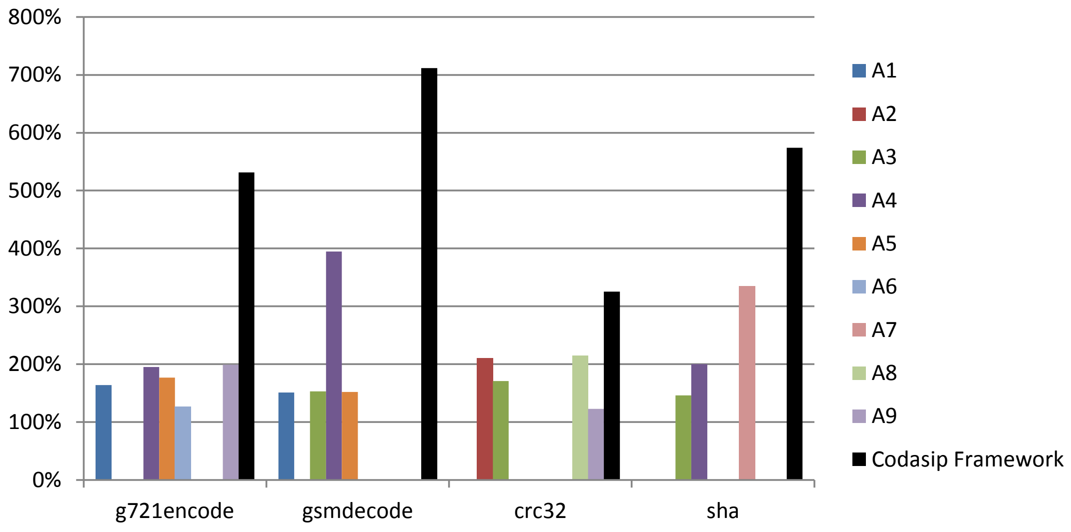


Figure D.2: Comparison of speed-ups for manually and automatically identified ISEs (1)

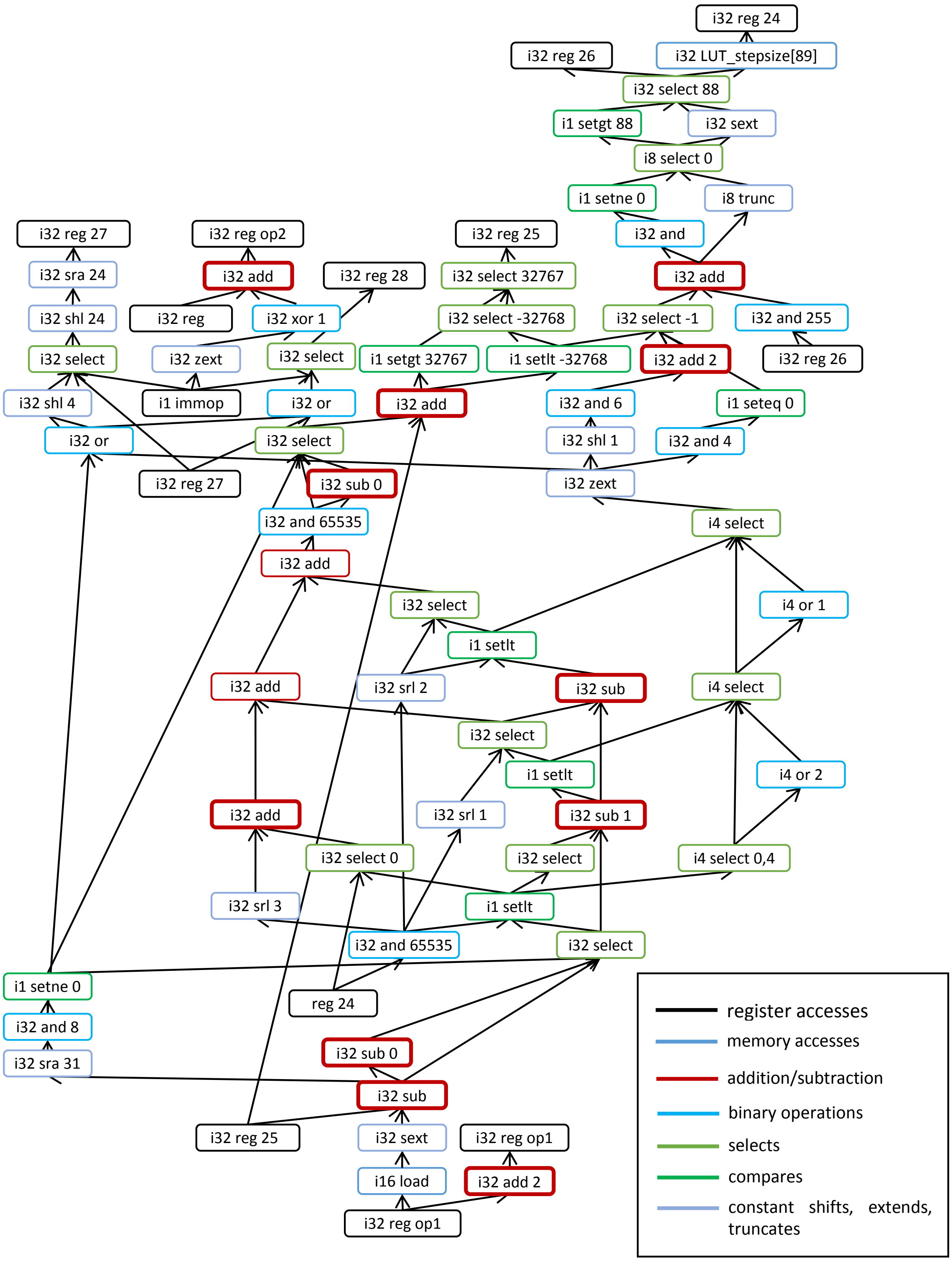


Figure D.3: Dataflow graph of an ISE for the MiBench rawcaudio benchmark

words in memory simultaneously. For the cache or main memory the multiple ports are too expensive, but for small dedicated memories the performance-area tradeoff may be interesting. The fifth freature is the automatic usage of lookup tables. Even small look-up tables that are used internally by an ISE can provide high speed-up. And finally, the sixth feature is related to shortening the ISE datapath. When the ISE input data signals are stable for several cycles and the ISE datapath is not pipelined, then outputs stabilize in several cycles. This allows lower latency at the cost of not being able to pipeline computations through the ISE datapath. According to the benchmarks analyzed in this article, such a behavior is rarely needed since many computations performed by ISEs need as input the results of the immediately preceding ISE.

Another optimization the author made as part of the evaluation of the Codix RISC processor core made by the Exar Corporation was the optimization for the AES Crypt application version 3.8.2. Three variants of the instruction set extension for the AES encryption were explored and one of these variants was chosen for hardware implementation. The speed-ups reported are calculated from cycles spent in the function `aes_encrypt`. For this optimization the Codasip Framework version 2.1.7 was used.

	Speedup	**Cost**
Variant 1	2.26 x	3x 32-bit 4-input MUX 2x 32-bit 2-input MUX 2x 32-bit XOR 1x 8-bit AND
Variant 2	7.38 x	1x 32-bit x 256 items lookup table with 4 reading ports 8x 32-bit 2-input MUX 4x 32-bit 4-input MUX 4x 32-bit XOR 2x 32-bit interstage registers
Variant 3	16.69 x	Vector register file with 4x 128-bit registers, 2 reading and 1 writing port The vector register file must allow writing the lower or higher 64 bits of a reg. The data cache or data memory must allow writing and reading consecutive 64 bits from address aligned to 8 bytes. 1x 8-bits x 256 items lookup table with 16 reading ports 16x GF256 multiplier by a constant 2 (multiplication in galois field) 16x GF256 multiplier by a constant 3 48x 8-bit XOR 5x 128-bit 2-input MUX 2x 128-bit 4-input MUX 2x 128-bit XOR 4x 32-bit 2-input MUX 2x 128-bit interstage registers

Table D.1: Speedup for AES Encryption on Instruction Accurate Simulator

Variant 2 of the extensions was implemented also in the cycle accurate model by Marián Pristach. An automatically generated RTL representation was then synthesized with Xilinx ISE 14.7.

Speed-up was measured as the time spent by repeatedly calculating the AES encryption with the original code divided by the time spent by repeatedly calculating the AES encryption with the optimized code. The application ran on the Linux operating system running on Codix RISC. Time was measured using the function clock.

The time that was measured also covers the execution of a loop and calling the `aes_encrypt` function. Therefore the speed-up is slightly lower than reported for the instruction accurate simulator, where only cycles spent in the `aes_encrypt` function were taken into account.

	Speedup	**Notes**
Variant 2	7.10x	This is the highest speedup obtained for an input data size of 8000 bytes.

Table D.2: Speedup for AES Encryption on FPGA

The highest speed-up was obtained for an input size of 8000 bytes because there is also an output data array of the same size and both the arrays fit the data cache whose size is 16kB. Also, the overhead of the loop that calls the `aes_encrypt` function was the smallest for this size. Even with larger input and output arrays of 15000 bytes in size that do not fit the data cache and cause data cache eviction and misses, the speed-up was 5.97 x.

Appendix E

Other Uses of Extracted Semantics

The instruction set model in the form produced by the Semantics Extractor is successfully used also for other purposes, including fast instruction set simulators, retargetable decompilation or test generation, and in this appendix, we will briefly overview each of these tools.

Just-in-time Compiling Simulator Generation

One of the essential tools for processor design, debugging and programming is the instruction set simulator. Codasip Framework provides an interpreting simulator for instruction accurate models. The interpreting simulator can simulate between 2 and 60 million instructions per second (MIPS) on a standard PC. Originally a compiled simulator was also provided that could simulate up to 100 MIPS [91], [90].

However, for some applications and to periodically run a large testsuite, such speeds are not sufficient, because, for example, the LLVM testsuite with the interpreting simulator ran for more than 40 hours on a standard PC and many tests failed the timeout. For such purposes, a faster simulator is necessary, even if it has not such good debugging capabilities.

Two platforms QEMU [92] and OVP [48], were chosen and generators of architecture definitions for these simulation platforms were implemented as a master's thesis at FIT BUT [77]. By using the instruction semantics as the input, it was possible to generate decode functions for QEMU, and morphing rules for OVP that translate each simulated basic block into the native code. This is then used by the Just-In-Time simulation [98] eengines of the QEMU and OVP platforms. The generated QEMU simulator speed ranged from 270-2200 MIPS, and the OVP simulator reached speeds between 680-2300 MIPS. The OVP and QEMU-based simulators are approximately 20-100 times faster than the original interpreting simulator. This substantially reduces the time needed for automatic C compiler testing and also allows simulating large applications whose simulation would be too slow with an interpreting simulator.

Decompilation

Another use of the instruction semantics is in reverse compilation. A binary object file serves as the input to the decompiler and then via a set of transformations, the C language or the Python code is generated (e.g. [58]). The instruction semantics is here used to transform the binary. Semantic models of x86, ARM, MIPS and PowerPC architectures are currently used by the decompiler. The decompiler is used by the AVG antivirus company to analyze suspicious binary code.

Verification

The instruction semantics was also successfully used in a processor verification tool, where the extracted semantics obtained from an instruction accurate (IA) CodAL model was converted to a representation in the SMV formal modeling language and an internal RTL representation of a Cycle Accurate (CA) model was converted to the same language. Then a SAT solver was able to formally check whether each instruction from the IA model was equivalent to what was happening in a CA model when the instruction had been executed [15].