TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Programme in Information Technology

VELI-PEKKA JÄÄSKELÄINEN

RETARGETABLE COMPILER BACKEND FOR TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Examiners: Prof. Hannu-Matti Järvinen
and Prof. Jarmo Takala
Examiners and subject approved by
Department Council
October 15 2007

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Programme in Information Technology
Jääskeläinen, Veli-Pekka: Retargetable Compiler Backend for Transport Triggered Architectures
Master of Science Thesis: 57 pages
March 2010
Major subject: Software Engineering
Examiners: Prof. Hannu-Matti Järvinen and Prof. Jarmo Takala
Keywords: transport triggered architecture, compiler

Embedded computer systems can be found everywhere as the result of the need to develop ever more intelligent and complex electronic devices. To meet requirements for factors such as power consumpiton and performance these systems often require customized processors which are optimized for a specific application. However, designing an application specific processor can be time-consuming and costly, and therefore the toolset used for processor design has an important role.

TTA Codesign Environment (TCE) is a semi-automated toolset developed at the Tampere University of Technology for designing processors based on an easily customizable Transport Triggered Architecture (TTA) processor architecture template. The toolset provides a complete co-design toolchain from program source code to synthesizable hardware design and program binaries.

One of the most important tools in the toolchain is the compiler. The compiler is required to adapt to customized target architectures and to utilize the available processor resources as efficiently as possible and still produce programs with correct behavior. The compiler is therefore the most complicated and challenging tool to design in the toolset.

The work completed for this thesis consists of the design, implementation and verification of a retargetable compiler backend for the TCE project. This thesis describes the role of the compiler in the toolchain and presents the design of the implemented compiler backend. In addition, the methods and benchmark results of the compiler verification are presented.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
Jääskeläinen, Veli-Pekka: Retargetable Compiler Backend for Transport Triggered Architectures
Diplomityö: 57 sivua
Maaliskuu 2010
Pääaine: Ohjelmistotuotanto
Tarkastajat: prof. Hannu-Matti Järvinen ja prof. Jarmo Takala
Avainsanat: transport triggered architecture, compiler

Seurauksena tarpeesta kehittää yhä älykkäämpiä ja monimutkaisempia laitteita, sulautettuja tietokonejärjestelmiä on nykyään kaikkialla. Nämä järjestelmät vaativat usein käyttötarkoitusta varten optimoituja mikroprosessoreita, jotta esimerkiksi virrankulutukseen ja suorituskykyyn liittyvät vaatimukset saataisiin täytettyä. Sovelluskohtaisten prossessoreiden suunnittelu voi kuitenkin olla aikaa vievää ja kallista, joten prosessorien suunniteluun käytetyllä ohjelmistolla on tärkeä rooli.

TTA Codesign Environment (TCE) on Tampereen teknillisellä yliopistolla kehitetty kokoelma ohjelmistotyökaluja, joka perustuu helposti muokattavaan "transport triggered architecture" (TTA) -suoritinarkkitehtuurimalliin. TCE:n työkalut tarjoavat puoliautomatisoidun prosessoreiden suunniteluvuon alkaen ohjelmien lähdekoodista päätyen syntetisoitavaan prosessorikuvaukseen ja prosessorilla suoritettavaan binäärimuotoiseen ohjelmaan.

Yksi tärkeimmistä suunniteluvuon työkaluista on kääntäjä. Kääntäjän on mukauduttava räätälöityyn kohdearkkitehtuuriin, käytettävä prosessorin resursseja mahdollisimman tehokkaasti hyväkseen ja tuotettava ohjelma, jonka toiminta on oikea. Tämän takia kääntäjä on TCE:n monimutkaisin ja toteutukseltaan haastavin työkalu.

Tämän diplomityön taustalla oleva työ koostui valmiiseen kääntäjäympäristöön suunnitellusta ja toteutetusta TCE-kohtaisesta moduulista, sen testaamisesta ja oikean toiminnan varmentamisesta. Diplomityössä esitellään kääntäjän rooli TCE:n työkaluketjussa ja kuvataan toteutetun kääntäjämoduulin arkkitehtuuri. Lisäksi diplomityössä kuvataan kääntäjän testauksessa ja suorituskyvyn mittauksessa käytetyt menetelmät tuloksineen.

PREFACE

The work for this thesis was done in the Department of Computer Systems at Tampere University of Technology as a part of the Flexible Design Methodologies for Application Specific Processors (FlexASP) project.

I would like to thank prof. Jarmo Takala for giving me the chance to work on an interesting and challenging project, which was a great learning experience and gave me a good perspective of the software and hardware development as a whole. I am also very grateful for Pekka Jääskeläinen, M.Sc., Pertti Kellomäki, Dr.Tech., and Prof. Hannu-Matti Järvinen for their invaluable feedback and ideas on how to improve my work. I would also like to thank my coworkers at the TCE project for creating an excellent working atmosphere and giving me the helping hand whenever it was needed.

Finally, I would like to thank my friends and family for their support throughout my studies and life.

Tampere, February 11 2010

Veli-Pekka Jääskeläinen

CONTENTS

LIST OF ABBREVIATIONS

ABI	Application Binary Interface
ADF	Architecture Definition File
AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
ASP	Application Specific Processor
BEM	Binary Encoding Map
CFG	Control Flow Graph
CU	Control Unit
DAG	Directed Acyclic Graph
DFG	Data Flow Graph
FU	Function Unit
GCC	GNU Compiler Collection
GPP	General Purpose Processor
GPR	General Purpose Register
HLL	High Level Language
HDL	Hardware Description Language
ILP	Instruction Level Parallelism
IR	Intermediate Representation
IU	Immediate Unit
JIT	Just-In-Time Compilation
JPEG	Joint Photographic Experts Group
LLVM	Low Level Virtual Machine
MOM	Machine Object Model
OSAL	Operation Set Abstraction Layer

PIG	Program Image Generator
POM	Program Object Model
PSNR	Peak Signal-to-Noise Ratio
RF	Register File
RISC	Reduced Instruction Set Computer
SSA	Static Single Assignment Form
TCE	TTA-Based Codesign Environment
TPEF	TTA Program Exchange Format
TTA	Transport Triggered Architecture
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word
XML	Extensible Markup Language

1. INTRODUCTION

Processors for embedded systems often have strict requirements limiting their design. Factors such as power consumption, performance, and production costs place much stronger restrictions for the processor architecture when compared to, for example, general-purpose processors (GPPs) in desktop computers. However, embedded systems are typically required to run only a very limited set of programs, allowing the processor design to be optimized for the application.

Application Specific Processors (ASPs) are processors that are customized for executing specific software. They can, therefore, be much more effective solutions for embedded systems than GPPs, but still retain more flexibility than an Application Specific Integrated Circuit (ASIC) designed for only one task. When an ASP is designed, the hardware and software parts of the system are developed simultaneously by codesigning the processor and software to benefit from the tailoring of the system for the specific application. The instruction set of an ASP is customized by removing any instructions that are not needed by the software and by adding custom instructions that increase performance of the system.

Designing an application specific processor is a demanding and time consuming task. A software toolset for designing Transport Triggered Architecture (TTA) ASPs called TTA-based Codesign Environment (TCE) was implemented at Tampere University of Technology to challenge this problem. TTA is a processor design paradigm for designing ASPs based on a modular architecture template, which allows customization of a processor by adding and removing basic building blocks of the architecture template such as function units, register files, and transport buses. TCE provides a full toolset containing all tools required to codesign a TTA processor and software for it.

A crucial part of the toolset is the compiler. Implementing a compiler for a customizable architecture template is an especially demanding task because the compiler has to adapt to the available resources in the customized target processor. The TCE compiler is implemented as a backend for the *Low Level Virtual Machine* (LLVM) compiler framework which provides the high-level language frontends and target independent analysis and optimization components. This thesis describes the requirements, design, and verification of a LLVM compiler backend for the TCE toolset.

The thesis is divided into the following chapters. Chapter 2 introduces the TTA processor architecture and describes the method of programming TTAs. The chapter also presents the TTA Codesign Environment and describes the role of the compiler in the toolset. Chapter 3 describes the basic structure of typical compilers and introduces basic compiler concepts. In addition, the chapter gives an overview of the LLVM compiler infrastructure and backend framework. The design and implementation of the TCE compiler backend is presented in Chapter 4. Chapter 5 contains the compiler test results and describes the testing methods used for verification and benchmarking. Chapter 6 concludes the thesis.

2. CODESIGN ENVIRONMENT FOR APPLICATION SPECIFIC PROCESSORS

This chapter gives a brief overview of application specific processors (ASPs) and their design process. In addition, the Transport Triggered Architecture (TTA) processor architecture paradigm for designing ASPs is introduced. This chapter also introduces the TTA Codesign Environment (TCE), which is a toolset for designing TTA processors.

2.1 Application Specific Processor Design

Application Specific Processors (ASPs) are processors which are custom designed to run a specific set of software efficiently. When the target application is well defined with limited functionality, a processor can be designed with resources that are customized to benefit the application. For example, any unnecessary operations can be removed from the operation set and highly specialized custom operations can be added to improve performance. This way the power consumption, chip area, and manufacturing costs can be minimized while still fulfilling the program execution speed requirements.

ASPs can have significantly better efficiency when compared to using general purpose processors for the same task, but as a trade-off the design process can be time-consuming and costly. Therefore the choice of an ASP architecture with a good toolset to assist and automate the design process is important.

In order to design an ASP, a toolset is required for modeling, evaluating, and compiling software for the customized processor architecture. Each tool in the toolset must be able to adapt to different architecture variations and, therefore, the architecture design space has to be limited to an architecture template. The template describes general characteristics of the processor architecture, but can allow customization of different resources, such as the instruction set, register files and the interconnection network.

When an architecture template is used, processors can be designed by means of design space exploration. Design space exploration is a process where a processor is designed iteratively. First, an initial architecture is designed and evaluated with a simulator. Based on the evaluation results, the design is then improved and re-evaluated until a satisfactory design is found. The process can be done manually

or by using varying degrees of automation with a toolset capable of improving and evaluating architectures automatically.

2.2 Transport Triggered Architecture

An important way to improve program execution speed is to take advantage of Instruction Level Parallelism (ILP). ILP is a term for the fine grained independency of operations, which allows multiple operations to be executed simultaneously. Superscalar processor architectures, such as modern desktop CPUs exploit ILP within the processor hardware by detecting the operation dependencies during run time. Operation execution order can then be reorganized, and multiple operations scheduled to be executed in parallel. However, this approach requires additional logic in the processor hardware to detect operation dependencies and allocating processor resources for the executed operations.

Very Long Instruction Word (VLIW) is a processor architecture which utilizes ILP by parallelizing instructions at compile time [1]. In VLIW architecture one instruction consists of multiple operations which are statically scheduled to processor execution units by the compiler. Processor hardware is therefore freed from the dependency detection logic, reducing the hardware complexity and lowering power consumption. In VLIWs, the interconnection network between execution units and other processor resources needs to be designed for all possible concurrent data transports. The growing complexity of the interconnection network therefore limits the scalability of VLIWs.

Transport Triggered Architecture is a processor architecture template similar to the VLIW architecture. TTA takes the VLIW idea of moving complexity from the hardware to the compiler even further, by also assigning data paths used by instructions at compile time. This is done by programming the individual data transports between processor components instead of the traditional approach of programming whole operations. In TTAs, the operations are executed as side effects of the data transports. Because the data paths are assigned at compile time and the compiler is aware of the limitations, the interconnection network can be kept relatively simple when new resources are added. [2]

The static scheduling of data transports has some drawbacks when compared to superscalar architectures. The code density is lower, because more bits are required to encode data transports to a TTA instruction than encoding an operation in a traditional operation triggered architecture. Additionally, TTA performance is very dependent on the compiler quality, and since all ILP logic is in the compiler it can become very complex.[3]

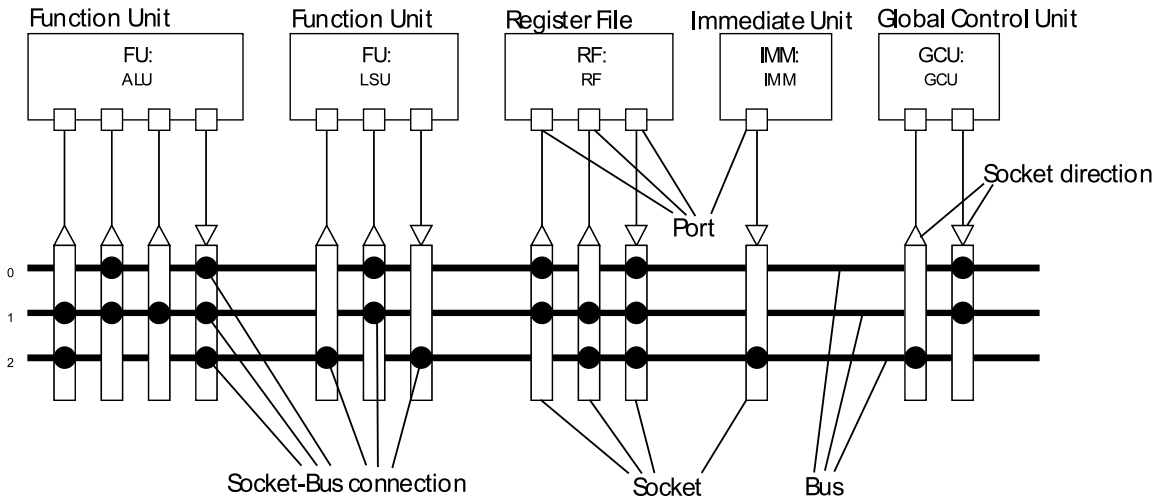

Figure 2.1: Module diagram of a simple TTA processor.

TTA Processor Organization. One of the main goals of TTA is to allow easy customization of processors with a templated architecture design. TTAs are built from components that can have pre-existing hardware implementation. Components called sockets and buses form the interconnection network which connects four different types independent units providing resources for operation execution. These unit types are function unit (FU), register file (RF), immediate unit (IU) and control unit (CU). Figure 2.1 shows a component diagram of a simple TTA processor with two function units, one register file, one immediate unit and a control unit.

Function units are the execution units of a TTA processor. One function unit contains logic to execute one or more operations. These operations can be simple operations such as addition of integers, or more complex operations that do computation specific to the application that the architecture is customized for. Usually at least one of the function units in a TTA processor is a special load and store unit that can access data memory. Function units read and write operands using input and output ports that are connected to the interconnection network. The operation set of a TTA processor can be customized by adding function units that provide the desired set of operations.

Register files are units that contain arrays of registers with same bit width. Registers are used for storing temporary values such as operation operands for fast access inside the processor. Registers are also used for special purposes such as storing the stack pointer and function return value. A TTA processor can have multiple register files with different bit widths. Register files can have multiple input and output ports to allow multiple register accesses in one instruction cycle.

Immediate units contain special registers to store long constant values that can not be encoded in instructions as literal constant values.

A TTA processor contains a control unit which is responsible for controlling the

processor operation. It fetches and decodes instructions and generates signals to execute them. The control unit also contains control flow operations so it can also be seen as a special function unit.

Sockets and buses form the interconnection network that is utilized to transport data between the units. The number of buses limits the number of concurrent data transports in a TTA processor. One bus can complete one data transport in each instruction cycle between unit ports that are connected to it by sockets. Sockets have a direction which determines if the ports connected to it can read or write the transport bus. Connections between sockets and buses are usually optimized to contain only the connections that are needed for fast processor operation. A fully connected interconnection network with multiple buses connected to all ports will usually have poor utilization and high cost in terms of chip area.

2.3 TTA Processor Programming

Traditional processors are programmed by defining operations and their operands. For example, in the assembly language of a traditional RISC architecture, simple operation like addition of two operands in registers r1 and r2 to register r3 might look like this:

```
add $r3, $r1, $r2
```

The RISC processor will generate the required signals to execute the operation. In contrast, TTA processors are programmed by defining the data transports that are required to perform the desired behavior. The actual operations are executed as side effects of the data transports when a data transport occurs to an operation triggering port of a function unit.

The same example in TTA assembly would look something like this:

```
r1 -> add.1
r2 -> add.2
add.3 -> r3
```

To execute the add operation, three data transports called *moves* are defined. The first move defines a data transport from register r1 to the input port 1 of a function unit containing the add operation. The second move defines another move from register r2 to the input port 2 which triggers the execution of the add operation. Finally, the result is moved from the output port of the function unit to the register r3. This example is *sequential* TTA code. Operation latencies are not yet taken to account, the moves are not parallelized and the required target resources are not yet assigned.

To assemble a real TTA program for a specific target TTA processor, the program must be *scheduled* for the target architecture and the processor resources must be allocated. Registers must be bound to specific registers in the register files of the target TTA. Operations must be also bound to specific function units of the target architecture containing the corresponding operations. Finally, the moves must be scheduled while considering operation latencies and parallelized to specific transport buses to exploit ILP.

Scheduling TTA programs for target architecture. The following is a small sequential TTA program with a conditional jump at the end. The syntax "!bool" in move 12 denotes that the move is conditional and will occur only if the value in the register "bool" is FALSE (binary value zero). This program also contains constant literal values which can be encoded in the instructions as short immediates.

```
1:     1 -> r1                    [initialize variables in registers]
2:     0 -> r2
3:     r1 -> add.1
4:     r2 -> add.2
5:     add.3 -> r2
6:     r1 -> eq.1
7:     1024 -> eq.2
8:     eq.3 -> bool               [bool = boolean register]
9:     r1 -> add.1
10:    1 -> add.2
11:    add.3 -> r1
12:    !bool 3 -> jump.1          [jump to 3: if bool equals zero]
```

The following TTA program is what this example might look like after allocating resources and scheduling moves to the buses of a target TTA. This *parallel* TTA program is target-dependent and can be assembled only for the target TTA it was parallelized for.

```
1:     1 -> RF1.1                 0 -> RF1.2
2:     RF1.1 -> FU1.add.i1        RF1.2 -> FU1.add.i2
3:     FU1.add.o1 -> RF1.2        RF1.1 -> FU2.eq.i1
4:     1024 -> FU2.eq.i2          1 -> FU1.add.i1
5:     FU2.eq.o1 -> RF2.1         RF1.2 -> FU1.add.i2
6:     FU1.add.o1 -> RF1.1        !RF2.1 2 -> GCU.jump.1
```

Registers r1 and r2 are allocated to registers 1 and 2 in a general purpose register file RF1. Boolean register bool is allocated to register 1 of a 1-bit register file RF2.

Operations are also bound to function units containing the corresponding operations, and the operand ports are specified to correspond the correct operand ports of the operations. Moves are then scheduled to two buses of the target TTA.

The conditional move in instruction 6 is done by utilizing a register guard on the second bus. This bus must therefore have a guard that can execute the move conditionally depending on the value of register 1 in RF2.

The instruction scheduler in a TTA compiler must also take operation latencies into account. Results can be read only when the function unit has done the calculation and the result is ready. Operation latency determines the number of cycles required by the function unit to compute the result of the operation.

2.4 TTA Codesign Environment

TTA-based Codesign Environment (TCE) is a set of tools developed at Tampere University of Technology for designing and programming TTA processors. The goal of the TCE project is to provide an easy to use toolchain for TTA processor design, aiming to minimize the time and cost of design by automating the design process as much as possible [4].

The TCE processor design toolchain contains all tools required to design and simulate TTA processors and programs. The processor design starting point is usually the source code for an application, and a set of performance requirements and design limitations that must be met by the designed processor. The processor is designed with an iterative *design space exploration* process. The exploration begins with an initial architecture acting as a starting point for the processor design. The program source code is compiled for the processor architecture, and a model of the compiled program is simulated on a simulator for the processor architecture. The simulator produces a trace of the program execution, which is examined to improve the architecture design. This process is repeated until a satisfactory design is found.

The design space exploration can be done either manually, or at different levels of automation. In the manual design process, the different tools of the processor design toolchain are used manually, and the processor design is modified by hand. The fully automatic design space exploration has only the program source code and the design requirements as input, and does not require any user interaction for the exploration process. A user's guide to different TCE tools and the design process can be found in the TCE User Manual [5].

2.4.1 Automatic Design Space Exploration

The automated design space exploration is driven by the *Explorer* tool. The Explorer is a highly modular tool which can be programmed to explore the design space

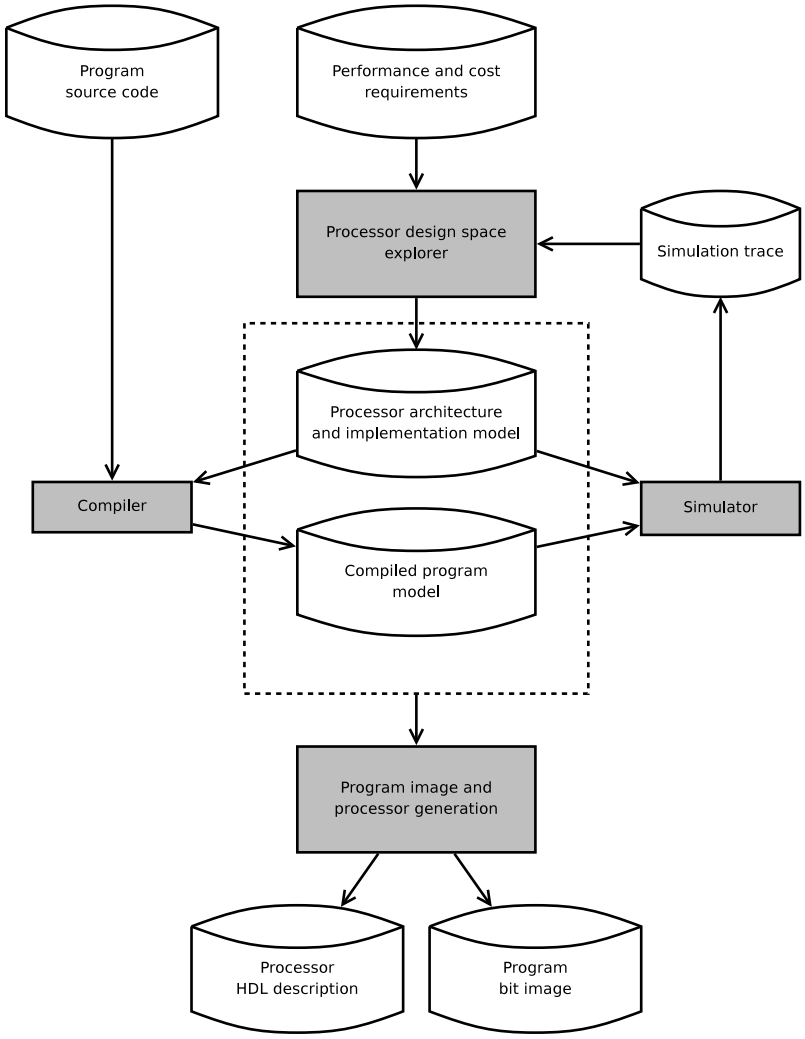

Figure 2.2: TCE design flow.

according to any user criteria. The design space explorer works in conjunction with the TCE compiler and simulator, which are used to test different architecture configurations. The explorer itself is responsible for estimating the cost of the different configurations, and modifying the architecture to different points of the design space in order to find an improved design. The design cost is measured in terms of the chip area required to implement the processor, and the total energy consumption to run the desired program. The processor performance is measured as the number of clock cycles required to run the program. The automated exploration design flow is illustrated in Figure 2.2.

The exploration begins with an initial processor architecture model, which can be for example the minimal required configuration needed by the compiler to compile arbitrary programs. The source program is compiled for the initial architecture. The compiler produces a model of a parallel TTA program which can be simulated. The simulator is invoked to simulate the parallel program model on a model of the

processor architecture. The simulator result is a simulation trace database. The trace contains detailed information about the program simulation, such as the total clock cycle count of the simulation and the utilization of different architecture components. The explorer will then generate a new configuration point in the explored design space which is tested. The process of exploring different configurations in the design space is repeated until the user defined criteria is fulfilled.

In order to estimate the power consumption and chip area, the explorer has to generate an implementation model of the processor. The implementation is generated by choosing implementations for the architecture components from a database of pre-existing hardware components with known characteristics.

When the final configuration is found, it can be prepared for implementation. TCE includes tools for generating a HDL description of the processor from the processor architecture and implementation models. The processor implementation model is also used for generating a bit image of the compiled program model.

2.4.2 Compiler

The TCE compiler is the subject of this thesis. The complier is a retargetable code generator, which can adapt to different architectures designed with the TCE architecture template. It gets the program source code and a processor architecture model as input. The output is a model of a parallel program for the target processor architecture. The design and implementation of the compiler is discussed in Chapter 4.

2.4.3 Simulator

In order to verify and benchmark an architecture configuration, the program execution must be simulated on a software model of the processor. The toolset contains a processor simulator with two simulation engines for this purpose. The simulation engines are a cycle-accurate interpretive simulator, and a faster but less accurate compiled simulator.

The interpretive simulator simulates an architectural model of the processor. The simulation model contains only architectural components, which are visible to the programmer. However, the simulation is cycle-accurate, and all architectural components contain correct data on each cycle [6]. The interpretive simulator engine simulates an assembly-level model of the program, not the execution of an actual bit image of the program. The interpretive simulator is useful for verification of an architecture with clock cycle level tracing and debugging of the program execution.

The compiled simulator engine generates executable simulation code from a parallel program model compiled for the target processor [7]. Individual clock cycles and

architecture components are not simulated. The simulation code also has limited error detection capabilities compared to the interpretive simulation. The reduced simulation overhead results in much faster simulation, which is useful for quick benchmarking of a target architecture.

2.4.4 Program Image and Processor Generation

The final step in the TCE toolchain is the generation of a processor implementation description which can be synthesized for the chosen hardware technology, and the generation of a program bit image which can be executed on the processor.

The processor implementation description is generated from the architecture and implementation models of the processor with the *Processor Generator* [8] tool. The tool produces a VHDL hardware description of the processor.

In order to generate a program image, a *Binary Encoding Map* (BEM) is generated for the processor implementation. The BEM contains the information required to encode instructions for the processor. A *Program Image Generator* (PIG) [8] tool uses the BEM to generate a bit image from a parallel program model compiled for the processor.

3. COMPILERS

A compiler is a software system that translates programs between source and target representations, typically converting programs written in a high level language (HLL) to a target machine specific representation. Compilers can be implemented using a compiler infrastructure which provides modular and reusable components for compiler implementation.

This chapter introduces general compiler concepts and the typical structure of compilers. In addition, the Low Level Virtual Machine (LLVM) compiler infrastructure, which is the basis of the TCE compiler is introduced.

3.1 Compiler Structure

Typically, compiler infrastructures have modular architectures where different programming language frontends and compilation target backends can be added as independent modules as illustrated in Figure 3.1. In order to achieve modularity, the compiler must have a well defined *Intermediate Representation* (IR) of programs that different modules use to communicate programs between compilation phases.

Intermediate representation. An IR of a compiler is a data structure that represents the compiled program to the compiler. Intermediate representation uses an intermediate language that targets an abstract target machine. The intermediate language consists of a virtual operation set of primitive operations. The virtual operation set and the structure of the IR are designed to aid in the analysis, optimization and code generation for target machines.

In generic compiler frameworks that are not designed for a specific target architecture, the intermediate language usually has a virtual instruction set of primitive operations that are generic to most processors. The virtual instruction set can also have abstract operations that are not specific to any machine, but represent a target-dependent operation sequence such as a function call or dynamic memory allocation for temporary values. The abstract representation of these operations allows high-level optimizations to be done in the compiler middle-end, but leaves the low level code generation the responsibility of the target specific backend.

The intermediate language operations usually perform operations on operands stored in virtual registers. The number of virtual registers available in the abstract

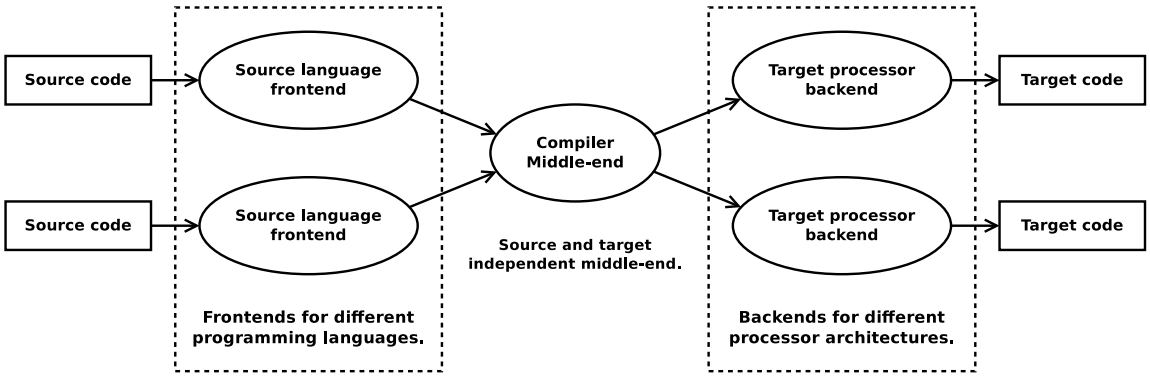

Figure 3.1: Structure of a typical compiler.

IR machine is usually very high or practically unlimited.

Compilers often use more than one form of intermediate representation. Different IRs are used for different phases of the compilation that are easier to perform on certain type of IR than another. For example, some optimizations and program analysis require a graph representation of the program where program operations and operands are represented as nodes of a tree graph exposing the program data flow. On another hand, some optimizations and compilation phases are much easier to implement using a *static single assignment form* (SSA) of the program. In SSA every variable is assigned only once, which simplifies the analysis of variables. If the same variable is assigned multiple times it is split to multiple versions that define a new variable. [9]

Frontend. A compiler frontend is responsible for parsing programs written in frontend-specific HLL and translating it to the IR of the compiler. In a standard organization, frontend structure is divided to three main parts [10]. First, lexical analysis is performed on the input data, which tokenizes the input string of characters to syntax specific tokens of the programming language structure, such as a keyword or a parenthesis. Next, the parser performs syntax analysis on the token stream, detects syntax errors and constructs a parse tree of the program. Finally, the semantic analyzer checks the program for static-semantic validity and an IR of the program is constructed. The IR produced by a frontend is usually source language independent.

Middle-end. The compiler middle-end is the source language and target machine independent part of the compiler. Middle-end is responsible for analyzing and optimizing the IR before passing it to a target backend. Compiler middle-ends usually perform data and control flow analysis on the IR. The resulting data flow graph (DFG) contains interdependencies between operations. Control flow graph (CFG) contains the basic block structure of the program and identifies control flow transfers

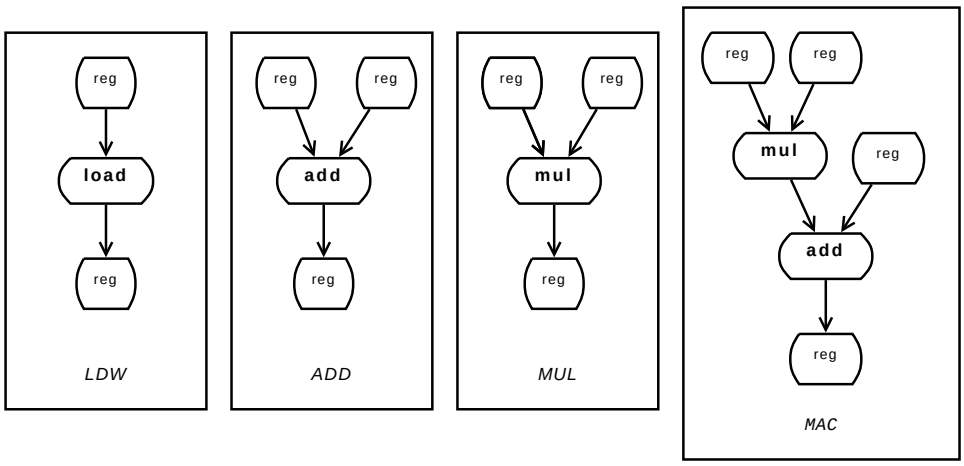

Figure 3.2: Four instruction patterns available for instruction selection.

between basic blocks. A basic block is a sequence of instructions that has a single entry and exit point, which can be treated as a single entity when the control flow of a program is analyzed.

These graphs are used as a basis for different optimizations done in the middle-end. Optimizations are usually implemented as modules, so that the sequence of optimizations can be customized. The resulting optimized IR is still target-independent, but the sequence of different optimizations can be chosen to benefit some specific target.

Backend. Compiler backends are target specific code generators, which translate the target independent IR to target specific code. The most important tasks for a backend is to perform *instruction selection*, *register allocation* and *instruction scheduling* on the IR. Backends can also perform target specific optimizations. The output of a compiler backend is usually assembly code or binary object code for the target machine.

3.1.1 Instruction Selection

An instruction selector maps IR operations to operations supported by the target processor. A single IR operation can be expanded to a sequence of target operations or an emulation function call if an IR operation is not included in the operation set of the target machine. In the opposite situation, the target machine might have operations that combine a sequence of IR operations to a single target operation.

Instruction selection is usually done on a DFG representation of the IR. An instruction selector has tree patterns of the target machine instructions defining the behavior of the instructions to perform instruction selection on a DFG. The patterns can also be associated with a cost. The task of instruction selector is to cover the

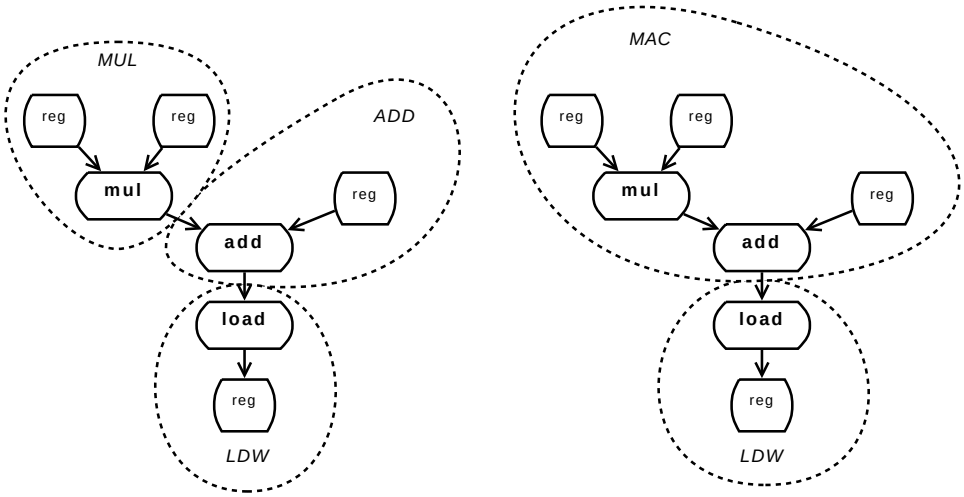

Figure 3.3: Two possible ways of covering a simple DFG with ADD, LDW, MAC and MUL patterns.

program DFG using the machine instruction patterns. There are usually multiple different ways to cover even a very simple DFG. In Figure 3.2 there are four different instruction patterns that are available to cover a simple subgraph of a program in Figure 3.3.

A simple instruction selection method is to rewrite the IR operation tree by matching subtrees with the instruction patterns of the target machine. In the example in Figure 3.3, the only available pattern to match the load operation subtree is the LDW pattern. The remaining subtrees are matched from bottom to up. There are two possible ways to cover the add operation as illustrated in Figure 3.3. Usually, an instruction selector would try to match a subtree starting from the pattern that covers the largest subtree. In this example, the MAC pattern is matched before the ADD pattern and the remaining graph is covered with a MAC operation. If the instruction selection patterns are associated with a cost, a linear time dynamic programming algorithm can be utilized to find the optimal solution [11].

3.1.2 Register Allocation

A register allocator maps the variables in the IR to real registers in target hardware. The virtual register utilization in the program is first analyzed to determine *live ranges* of the values in the registers. The live range of a value in register determines when it is safe to assign a register for a new variable. If the number of simultaneous live values in virtual registers is greater than the number of physical registers, the register allocator must *spill* values to memory. The register allocator can do this by inserting *spill code* which stores a live register value to memory. The register can then be used for a new variable, and the old variable can be restored later from memory when it is needed again.

Spilling register values causes load and store overhead, so the task of a register allocator is to minimize register spilling by maximizing the use of physical registers for frequently used variables. Register allocation is a computationally intensive NP-complete [10] problem and there is no known efficient algorithm to find the optimal solution.

3.1.3 Instruction Scheduling

Processors can be pipelined to improve their instruction throughput. An instruction pipeline splits the execution of an instruction to a sequence of independent steps. Multiple instructions can therefore be processed in parallel by executing different stages of instructions at the same time in the pipeline. However, pipelining introduces *hazards* to the computation when simultaneously executed instructions have interdependencies. Hazards are usually avoided by stalling the pipeline.

Instruction scheduling is a compiler optimization which reorganizes the execution order of instructions to avoid pipeline stalls for improving performance. Some machines also expose instruction pipeline resources to the compiler and expect the compiler to take care of operation timing issues by always scheduling instructions.

Instruction scheduling is especially important for VLIWs and TTAs. Both architectures have programmer visible operation latency that has to be taken into account by the compiler. Instruction scheduling is therefore always needed for VLIWs and TTAs to maintain correct behavior of programs. On TTAs, the quality of the instruction scheduling is also very important to the program execution performance.

In TTA processors individual function units can be pipelined with resources that are visible to the compiler. Other processor resources such as buses, guards and ports are also visible to TTA compilers. Instruction scheduling is therefore a more complex and important problem for TTA processors than for example superscalar architectures. A TTA instruction scheduler has to allocate processor resources to executed instructions and take operation latencies into account while packing maximum number of moves to instructions to improve ILP. More information on TTA instruction scheduling can be found in [12].

3.2 Compiler Retargeting

The easiest way to implement a compiler for a processor architecture is to implement a backend for an existing compiler infrastructure. Compiler infrastructures usually aim to minimize the manual work required for writing new backends. Backends are typically implemented using a framework which provides a formal machine description language. The purpose of machine description languages is to capture the compilation related details of the processor and allow reuse of generic backend

code.

Modeling languages vary on the approach they take to model a processor. Behavioral modeling languages such as ISDL are more compiler-oriented and allow easier implementation of the backend but require more knowledge of compilers. Structural modeling languages such as MIMOLA describe the structure and architectural details of the processors and are therefore easier for users that are not familiar with compilers. A good overview of customizable processor architecture description languages can be found in [13]. Depending on the compiler framework and the modeling language, backend generation can be fully automated from the modeling language description, or might require some parts of the backend to be manually programmed.

Static compiler backends are limited to a single target processor or a family of similar processors. The backends have hard-coded models of the target processor architecture, and exact properties of the processors must be known when implementing the backend. This poses a problem for customizable processor architecture templates where the processor is designed for a specific task. Implementing a new backend for each new architecture would be time-consuming and it would make fast processor prototyping and effective design space exploration impossible.

A solution presented in this thesis to this problem is to implement a dynamically retargetable backend, which only contains general aspects of the architecture template as a hard coded model. The actual target processor model is given as an input for the backend at runtime, and the backend adapts to the properties of the target processor configuration.

3.3 Application Binary Interface

A compiler backend is responsible for implementing code generation corresponding to the target specific *application binary interface* (ABI), which defines the low-level interface of program components.

An important aspect of an ABI is the memory organization of the target architecture. To achieve compatibility between different program components, the compiler backend has to implement a consistent memory allocation scheme defined in the ABI. A backend follows the ABI conventions, which define memory structure details such as the size and alignment of different data types in memory, and the structure of the call stack.

The call stack is a data structure which stores information about active functions in the program in stack frames. A stack frame has a consistent structure with predefined areas for function return address, function parameters, local variables and similar data specific to the state of an individual function invocation. This allows the function to access the local state data using a stack pointer, which points to the end address of the stack frame of the active function.

Another important part of the ABI is the calling convention. Calling convention determines how the function arguments are passed and how the return value is retrieved when a call to a function is invoked. The task of the backend is to convert the abstract call and return instructions to code sequences which implement the calling convention. On the caller side, a code sequence inserted to first store the function parameters to argument registers or the call stack, and another code sequence to retrieve the return value after the call has returned. On the callee side, the backend generates a function prologue to read the function arguments and an epilogue to store the return value in appropriate locations before returning from the function call.

The calling convention also defines the responsibility of saving and restoring registers which must preserve their values. Caller saved registers are saved to the stack by the calling function before the function call and restored after the call returns. Callee saved registers are saved in the called function prologue and restored in the epilogue before the function returns.

3.4 LLVM Compiler Infrastructure

Low Level Virtual Machine [14] is an open source compiler infrastructure based on low level virtual machine code representation of compiled programs. The goal of the LLVM project is to provide a robust platform for compiler development using a code representation which allows reuse of compiler components across different targets. LLVM has a modular design, allowing new language frontends, target machine backends, and optimizations to be easily incorporated in the existing framework. LLVM also aims to provide a flexible framework for program analysis and transformation, including compile-time, link-time, and run-time optimization and profiling [15]. It is therefore well suited for developing a compiler and researching new optimization methods for TTA processors.

LLVM is the compiler infrastructure chosen for TTA Codesign Environment (TCE). This section serves as a basis for the TCE compiler implementation presented in Chapter 4.

3.4.1 Compilation workflow

LLVM compilation workflow begins with compiler frontends for different source languages. Frontends emit bitcode which is linked together by a bitcode linker. The bitcode linker performs link-time optimizations, including inter-procedural analysis and optimization. The linked bitcode can then be optimized by an offline optimizer. Finally, a code generator writes the target assembly or binary machine code from the optimized IR.

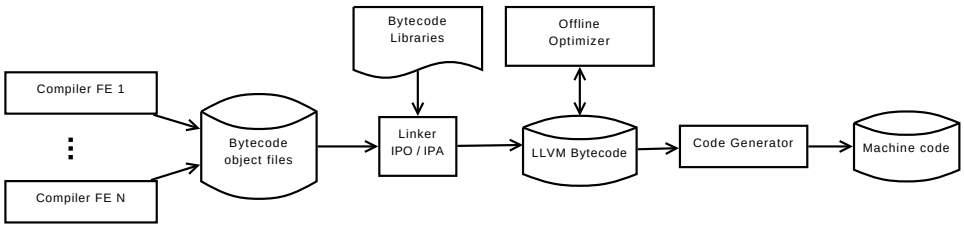

Figure 3.4: LLVM static compilation workflow.

LLVM also allows native execution of bitcode using Just-In-Time (JIT) compiler with runtime profiler and optimizer. However, the runtime optimizer and JIT cannot be used when LLVM is used as a static cross-compiler generating code for non-native machines. Figure 3.4 shows the static compilation workflow, when LLVM is used as a static cross-compiler.

The most popular of the current frontend implementations is a GNU Compiler Collection (GCC) [16] based LLVM-GCC frontend. The LLVM-GCC frontend includes support, among others, for C and C++ programming languages.

Optimization, analysis, and code generation are implemented as passes in the LLVM compiler. Each pass implements one analysis or transformation of the IR. LLVM has different types of passes working on different scopes, ranging from general module passes transforming or analyzing whole compilation units at a time to basic block passes that are limited to scope of a single basic block. Passes are organized with a *PassManager*, which manages pass dependencies and the execution order of passes. All passes use the LLVM IR as input and output.

3.4.2 Program representation

LLVM code representation is based on a source language and target machine independent RISC-like virtual instruction set. The virtual instruction set also includes instructions that expose high level language features to the compiler middle-end for effective optimization. LLVM code can be represented as human readable assembly language, as on-disk binary bitcode and as in-memory object model IR, which are all equivalent and use the same virtual machine as an intermediate target machine.

At a high level, LLVM programs are composed of modules which are the compilation units of the LLVM compiler. Modules consist of functions, global variables and symbol table entries. LLVM functions consist of basic blocks which in turn contain instructions. An instruction contains an opcode and a vector of operands.

The LLVM operands are strongly typed. The type system includes primitive types for fixed point integers with different bit-widths (i1, i8, i16, i32, ...), and floating point types of varying precision (f32 , f64, ...). The type system also contains more abstract types, such as code labels, pointers and different kinds of structured types.

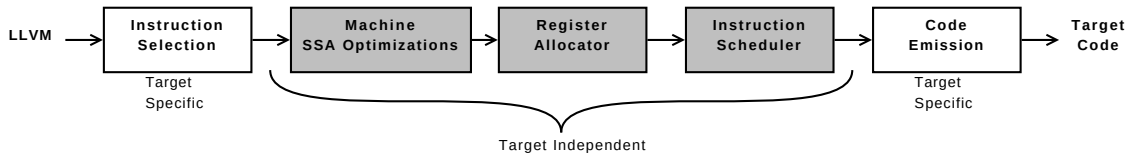

Figure 3.5: Basic LLVM code generator layout [18].

The primary representation of LLVM code is in Static Single Assignment (SSA) form [15]. The LLVM SSA form uses a virtual register set that contains an infinite number of typed virtual registers. Full specification of the LLVM language can be found in the LLVM Language Reference Manual [17].

3.4.3 Code generator

LLVM provides a framework for code generator implementation. The framework consists of multiple passes that have LLVM IR as input and target machine code as output. The high-level structure of a basic LLVM code generator is illustrated in Figure 3.5.

The first code generation phase is instruction selection. The instruction selection process starts with building of an initial instruction selection DAG, followed by DAG optimizations to simplify it. The selection DAG operand types are then legalized by transformations that convert any unsupported types to types which are supported by the target machine. After the legalize stage and additional optimizations, the selection DAG is ready for the actual instruction selection. The final stage of the instruction selection phase is formation of an SSA-representation of the intermediate machine code.

Instruction selection is followed by three phases, which have target independent implementations available in the LLVM framework. First, the machine code SSA can be optimized with optional SSA-based optimizations. The SSA is then register allocated, transforming the virtual registers to concrete registers in the target machine. LLVM libraries include multiple different built in register allocation algorithms that can be used. After register allocation, function prologue and epilogue code is inserted, the instructions are scheduled and late machine code optimizations are done to ready the program for code emission. The code emission is typically done by an assembly writer, which produces assembly code for the target processor.

3.5 Backend Implementation

Backends utilizing the LLVM code generator framework are implemented as target machines of the LLVM framework. The target machine interface allows custom implementation of all different code generation related passes. However, the code generator framework provides implementations for instruction selection and register allocation, which can be used by describing the related properties of the target machine using LLVM classes.

Target machine. Target machine encapsulates all properties and code generation methods of a target machine behind one interface. The LLVM code generator framework uses the interface to provide a backend for the target machine. Code generator passes are also created through this interface allowing full customization of the code generator.

Data layout. Data layout of a target machine describes the endianess and pointer size as well as sizes and alignments of different data types in the target machine memory.

Frame info. Frame info describes the basic properties of the target machine stack and stack frame layout. It also holds information about the direction of stack growth in the memory and stack frame alignment.

Register set. Target machine registers and register types are defined in the target machine register set. It describes all registers of the target machine and properties of different register types. The register set implements TargetRegisterInfo interface, which contains various methods utilized by the code generator to handle register and stack frame access. TargetRegisterInfo is also responsible for emitting prologue and epilogue code into functions.

Instruction set. Target machine instruction set contains descriptors of all supported instructions and defines their operand types. The operand types define different memory addressing modes and immediate operands. Register operand types are defined by referring to the register set. Instruction descriptors define the following properties of supported target instructions:

- Input and output operand lists, defining the operand types and number of operands,
- Instruction pattern which defines the behavior of the instruction as a DAG of LLVM instructions,

- Additional predicates and constraints for matching the instruction pattern,
- An assembly string template for printing the instruction as target assembly code,
- List of registers implicitly used and modified,
- Target-independent flags that define properties such as if the instruction is commutable or if it may read or write memory, and
- Target-specific flags and properties.

One target machine instruction requires typically more than one descriptor to describe the different combinations of operand types it can have and different instruction patterns it can be matched with.

Instruction selector. LLVM provides a base implementation for an instruction selector. A skeleton of the instruction selector is implemented as base classes in the LLVM code generator library and part of the instruction selector can be automatically generated from the instruction descriptors. Some of the instruction selector methods have to be manually implemented because the behavior of all instructions cannot be fully expressed with the instruction descriptors.

The manually implemented parts of the instruction selector include a legalize phase which converts unsupported types and operations to ones supported by the target machine. The legalize phase can either promote, expand, or implement a custom lowering for operations that are not supported. If an operation is supported for larger operand types, the unsupported operands can be promoted to supported types. Expand breaks an LLVM instruction to a combination of other instructions that perform the same operation. If promotion or expansion is not sufficient, the target legalizer can implement a custom code generation method for the unsupported operation.

The instruction selector also includes methods to support target calling conventions. Different calling conventions can be implemented by writing methods that generate code for passing arguments and handling return values of function calls.

Register allocator. LLVM code generator includes multiple register allocators that implement different register allocation algorithms as code generator passes. The register allocator implementations are target independent and do not require any target specific modifications. They utilize the target machine interface to access target register set model and use virtual code generation methods implemented in the target machine to generate code for spilling values to memory.

Target assembly information and assembly printer. LLVM contains a partial implementation of an assembly printer pass. The assembly printer can be utilized by providing information about the target assembly directives and by implementing required assembly printing methods manually. Part of the assembly printing methods can be automatically generated from the assembly string templates defined in the target instruction descriptors.

3.6 Target Descriptor Files

One of the goals of the LLVM project is to minimize the manual programming work required to implement a new backend. A backend has to model the target machine which includes repetitive descriptions of different target architecture resources. In order to make the target architecture modeling easier, LLVM includes a general purpose *TableGen* tool for processing records of domain-specific information. The main use of the TableGen tool is the LLVM code generator, which allows part of the backend C++ code to be generated from *target descriptor* files which contain records of a target machine properties. The target descriptor files can be processed by target description TableGen backends which generate code for essential parts of the backend.

The automatically generated code includes data structures that are required by the code generator framework to handle target machine instructions and registers, as well as methods for instruction selection and assembly code emission.

TableGen files consist of class and definition records. Classes are the abstract records that describe the structure of concrete records belonging to the same conceptual class. New TableGen classes can be derived from existing classes inheriting their properties. Definitions are the concrete records that define the properties of a domain-specific object. TableGen files can also contain multiclasses, which allow instantiation of group of classes in a single record resulting in multiple definitions.

The following part of this chapter is an introductory description of some of the key elements in typical target descriptor files. TableGen has a versatile and expressive syntax, which is not covered by this thesis. A comprehensive explanation of the TableGen syntax can be found in [19]. The TableGen target descriptor file backend is documented in [20].

Register is the base class for register records. It contains attributes defining the register properties and relationship to overlapping registers. The *Register* class defined in the target descriptor TableGen backend has the following structure:

```
class Register<string n> {
  string Namespace = "";
  string AsmName = n;
  int SpillSize = 0;
  int SpillAlignment = 0;
  list<Register> Aliases = [];
  list<Register> SubRegs = [];
  list<int> DwarfNumbers = [];
}
```

This base class can be subclassed to define complex target registers with minimal repetition as is done in the following example taken from the Sparc backend in LLVM:

```
class SparcReg<string n> : Register<n> {
  field bits<5> Num;
  let Namespace = "SP";
}

class Rd<bits<5> num, string n, list<Register> subregs> :
  SparcReg<n> {

  let Num = num;
  let SubRegs = subregs;
}
```

The first definition introduces a new class *SparcReg* which is derived from the Register base class. It adds a new 5-bit identifier field to store the register identifier number and assigns register instances to the *SP* namespace. This class is subclassed further with class *Rd* to define 64-bit slots in a floating point register file where each slot can consist of sub-registers. Finally, a 64-bit register *D0* consisting of two subregisters *F0* and *F1* can be defined with the following simple definition, giving arguments to the *Rd* class template:

```
def D0  : Rd< 0,  "F0", [F0,   F1]>, DwarfRegNum<[32]>;
```

The definition includes the Dwarf number of the register, which is used as an register identifier by debugging tools.

RegisterClass contains registers, which are grouped as members of the class. A register class record defines the supported types of the member registers, which must be the same for all registers of the class. The supported types are defined as LLVM IR value types. Register classes can be instantiated by defining a *RegisterClass* record. For example, the Sparc backend defines the following register class, which contains 64-bit registers, such as the *D0* register defined in the previous example:

```
def DFPRegs : RegisterClass<"SP", [f64], 64, [D0, D1, D2, D3,
  D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, D14, D15]>;
```

The first argument of the *RegisterClass* template is the namespace of the definition. The second argument is a list of LLVM value types that can be stored in the registers of this register class. The third argument is the alignment in bits that is required when loading or storing register values to memory. The last argument is a list of registers belonging to the class. The order of the list is used as the preferred register allocation order by LLVM register allocators.

Instruction is the target descriptor base class for defining the instruction records. The base class contains an extensive set of attributes to define behaviour of instructions. Similarly to register records, the *Instruction* class can be subclassed to allow groups of similar instructions to be expressed in a compact form. The following code is part of the *Instruction* base class definition. Most of the flags and attributes that define detailed properties of the instruction are omitted:

```
class Instruction {
  string Namespace = "";
  dag OutOperandList;
  dag InOperandList;
  string AsmString = "";
  list<dag> Pattern;
  list<Register> Uses = [];
  list<Register> Defs = [];
  list<Predicate> Predicates = [];
  bit mayLoad      = 0;
  bit mayStore     = 0;
  // Rest of attributes omitted
}
```

The input and output operand lists are defined as strings that represent DAGs that model the operands. The DAG representation allows complex operand types, such as operands of different memory addressing modes to be expressed as in the

operand list. Each operand is named so it can be referenced by the instruction patterns and assembly string.

The instruction patterns are also defined as DAGs. The patterns are used to generate code for the DAG to DAG instruction selector. The patterns are defined using predefined pattern fragments that correspond to different LLVM instruction nodes in a *selectionDAG*. The DAG fragments can be nested to express complex operation patterns. The DAG operands refer to operand records in a similar way as the input and output operand lists. The operands are mapped to the input and output lists by the operand names. The following example is a typical subclassed instruction record, taken from the Sparc backend:

```
def FDIVD  : F3_3<2, 0b110100, 0b001001110,
    (outs DFPRegs:$dst), (ins DFPRegs:$src1, DFPRegs:$src2),
    "fdivd $src1, $src2, $dst",
    [(set DFPRegs:$dst, (fdiv DFPRegs:$src1, DFPRegs:$src2))]>;
```

This record defines an instruction for double precision floating point division. The *F3_3* instruction format is subclass of the *Instruction* class. The template has six arguments that instantiate an instruction record. The first three arguments *2, 0b110100, 0b00100110* are values for Sparc specific instruction encoding fields. The fourth and fifth arguments *(outs DFPRegs:$dst), (ins DFPRegs:$src1, DFPRegs:$src2)* are the output and input operand lists. The lists define that the instruction has one input and two output operands stored in *DFPRegs* class registers. The lists also assign names *$dst*, *$src1* and *$src3* for the operands.

The sixth argument is an assembly string template used by the assembly printer. The template uses operand names defined in the input and output operand lists to create slots for the operand assembly strings.

The final argument is a pattern for the instruction, which uses two predefined pattern fragments. The *fdivd* fragment matches a selectionDAG node which corresponds to the LLVM double precision floating point division operation. The outermost *set* fragment makes the whole pattern match to a DAG where a result of a division of two DFPRegs values is stored to a DFPRegs class register.

Explicit selection DAG patterns. Typically, most instructions are selected using the instruction selection patterns defined in instruction records. However, in some cases, an explicit instruction selection pattern is required. The explicit selection DAG patterns match an instruction pattern, and produce a DAG of target machine instructions as a result. The patterns are defined using the following class template:

```
class Pattern<dag patternToMatch, list<dag> resultInstrs> {
  dag             PatternToMatch  = patternToMatch;
  list<dag>       ResultInstrs    = resultInstrs;
  list<Predicate> Predicates      = [];
  int             AddedComplexity = 0;
}
```

In most cases, only a simple result with one DAG is required. A pattern with one result DAG can be defined using the following subclass:

```
class Pat<dag pattern, dag result> : Pattern<pattern, [result]>;
```

The first template parameter is the matching pattern used in instruction selection. The second parameter is a DAG of target machine instructions which is the result of the instruction selection for the pattern. The following is an example of explicit selection DAG pattern, taken from the LLVM *Mips* backend.

```
def : Pat<(setge CPURegs:$lhs, CPURegs:$rhs),
          (XORi (SLT CPURegs:$lhs, CPURegs:$rhs), 1)>;
```

This selection DAG pattern matches greater-or-equal instructions, and selects them to a DAG which performs the operation with a less-than *SLT* operation followed by negation of the result with an exclusive-or *XOR* operation.

4. IMPLEMENTATION

The TCE compiler is based on the LLVM compiler infrastructure. Due to special requirements of TCE, the compiler is not implemented as an ordinary static LLVM compiler backend.

The most important requirement for the TCE compiler is retargetability. This poses a problem with LLVM compiler infrastructure where properties of the target machine are hard-coded in the backend. TCE requires a compiler which can adapt to templated architectures automatically without building a new compiler for each TTA.

Due to limited project resources, another requirement for the compiler implementation was to reuse LLVM libraries as efficiently as possible. This requirement ruled out a completely customized LLVM backend with TCE-specific dynamically retargetable instruction selector and register allocator.

LLVM backends are normally implemented directly into LLVM source code tree and built as a static part of a customized LLVM build. In TCE, the automatic design space exploration tools have to invoke the compiler within the program code. It is also desired that the TCE backend is kept as loosely tied to a specific version of LLVM as possible to ease portability to future versions of LLVM. For these reasons the TCE compiler is not implemented directly as a part of a LLVM build. Instead, the TCE compiler is implemented as a library in the TCE source tree which utilizes unmodified LLVM libraries and tools installed in the host system.

4.1 TCE Data Structures Used by the Compiler

This section presents the data structures and file formats of TCE which are used by the compiler.

4.1.1 Processor Architecture Model

The processor architecture template of TCE is modeled with the *Machine Object Model* (MOM) data structure [21]. A MOM instance defines the architectural layout of a TCE processor and contains all the information required to program the processor. The processor is modeled as architectural components such as function units, register files, and transport buses.

The instruction set of an architecture is visible through the function units in the

MOM. The operations in function units are references to *Operation Set Abstraction Layer* operation descriptions, described in Section 4.1.2.

The register file components in the MOM define the properties of all registers in the architecture. The compiler can use this information to build a list of available general purpose registers and their properties.

Machine object models can be serialized to XML-based *Architecture Definition Files* (ADF)s. ADF files are used to pass architecture models of processors between different tools of the TCE toolset.

4.1.2 Operation Set Abstraction Layer

Operation Set Abstraction Layer (OSAL) is a library for defining properties and semantics of operations in TCE machines. The operation properties are stored in XML-format files.

Each OSAL operation defines the name, the number of input operands, the number of output operands, and the operand types of the operation. OSAL operations also have additional attributes which define properties such as which input operands are commutable, and flags which tell if the operation may read or write memory. An operation may also include one or more data flow graphs defining the operation semantics with other OSAL operations.

TCE contains a database of basic operations, which are commonly used in the designed processors. New operations can be added by writing a description of the operation properties in a XML-file. The following is an example of a user defined *ANDN* operation:

```
<operation>
  <name>ANDN</name>
  <inputs>2</inputs>
  <outputs>1</outputs>
  <in id="1" type="UIntWord"/>
  <in id="2" type="UIntWord"/>
  <out id="3" type="UIntWord"/>
  <trigger-semantics>
      SimValue negResult;
      EXEC_OPERATION(not, IO(2), negResult);
      EXEC_OPERATION(and, IO(1), negResult, IO(3));
  </trigger-semantics>
</operation>
```

This operation description defines a combined bitwise *AND-NOT* operation. The operation is defined to have two unsigned integer input operands and one unsigned integer output operand.

The optional *trigger-semantics* section describes the operation semantics with existing OSAL operations. The compiler can use the semantics definition to automatically exploit a user defined operation during compilation. The semantics of the *ANDN* example define that the operation is interchangeable with a sequence of operations, where the second input operand bits are first negated with a *not* operation, and the negation result is then passed to an *and* operation with the first input operand.

4.1.3 Program Model

Program Object Model is an assembly-level intermediate representation of programs used by TCE tools [22]. POMs can represent parallel programs instruction scheduled for a target architecture, or sequential programs which are instruction selected and register allocated for a target architecture, but not yet scheduled. A POM consists of an object hierarchy representing the program code and data definitions which model the program and data memory contents.

The program code hierarchy consists of procedures, which contain basic blocks of instructions. An instruction in a parallel POM consists of moves which define the data transports for each transport bus of the target machine on the instruction cycle. A sequential POM instruction consists of a single move, which is not yet allocated to a specific transport bus or scheduled relative to other moves in the program.

The data definitions constitute the data memory contents and the symbol table of the data memory. Data definitions represent discrete objects in the memory, such as individual global variables or more complex data structures. A data definition may contain initialization data, which is used as the initial contents of the defined memory area. POMs can be read and written to binary *TTA Program Exchange Format* (TPEF) files to pass programs between different tools of the TCE toolset.

4.2 Minimum Target Machine Configuration

An important aspect of the compiler retargetability is the ability to compile arbitrary C language source code for target machine configurations with minimal resources. The main concerns from the code generation perspective are the minimum register file configuration and the minimum operation set.

The required number of registers is largely determined by the reserved registers needed for the calling convention and the registers required for execution of operations. The current implementation requires at least five 32-bit registers and two boolean predicate registers.

The minimal operation set consists of operations required by the instruction selector to match any LLVM instructions and the operations that can be inserted by

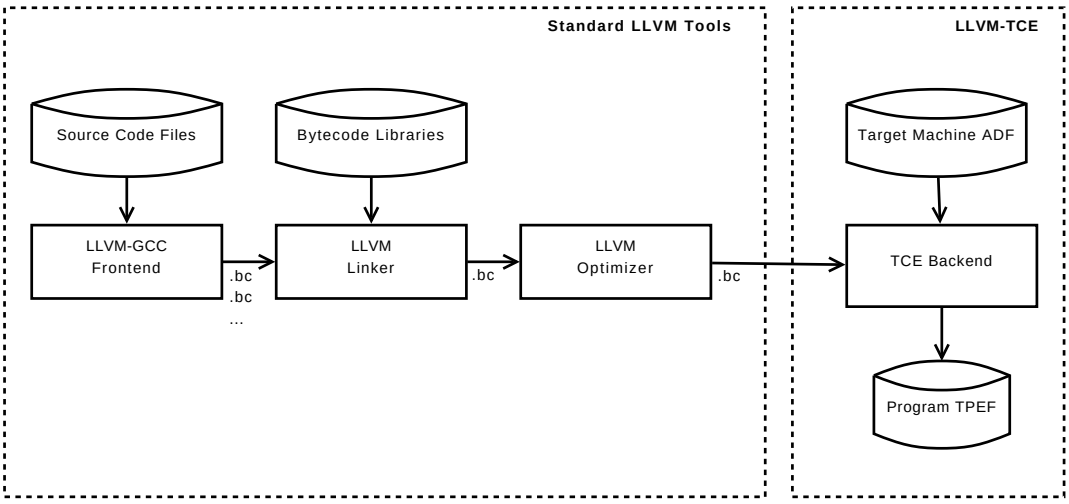

Figure 4.1: TCE Compiler Toolchain.

the code generator after instruction selection. Currently, the following operations are required:

- Addition (ADD) and subtraction (SUB) integer arithmetic operations.
- Greater than (GT), equal (EQ) and unsigned greater than (GTU) integer comparison operations.
- AND, inclusive-or (IOR) and exclusive-or (XOR) logical operations.
- Arithmetic bit shift operations to left and right.
- Logical bit shift to right operation.
- Load and store memory operations for word, half-word, and byte bit widths.
- CALL and JUMP control flow operations.

4.3 Compiler Toolchain

The TCE compiler toolchain consists of a compiler frontend, a set of target independent LLVM tools and a TCE code generator. The TCE code generator is implemented as a library which is part of the TCE project and separate from the LLVM tools. High level organization of the compiler toolchain is illustrated in Figure 4.1.

The first step in the toolchain is the LLVM-GCC frontend. LLVM-GCC is a modified version of gcc distributed with LLVM, used as a frontend to produce the initial bytecode. The LLVM-GCC frontend has a disadvantage of not being completely target independent. The frontend must be configured to output bytecode that is compatible with the target code generator. In practice this means that source

language data type sizes and endianess must be defined to be compatible with the target machine. The compiler toolchain includes a version of the LLVM-GCC frontend with target machine configuration for the TCE code generator.

The next steps of compilation are linking and optimization. Both steps are done with unmodified LLVM tools. The compiler expects these tools to be found in the host system and they are not included in the TCE compiler. All compiled code must be fully linked as one bytecode module before code generation because TCE does not currently support linking of binary machine code.

The final step in the compiler toolchain is the LLVM-TCE code generator. LLVM-TCE is a TCE backend implementation for LLVM. Unlike traditional LLVM backends, LLVM-TCE is implemented as a stand-alone code generator which utilizes LLVM code generation libraries instead of being implemented as a static part of the LLVM compiler. The code generator has two input files: a fully linked and optimized bytecode of the compiled program and the architecture definition of the target machine. The backend is able to dynamically retarget itself to the target machine without the need of recompiling the whole code generator.

The whole toolchain can be invoked from command line using the *tcecc* compiler driver script. The script aims to be compatible with commonly used gcc compiler switches to minimize work when porting build scripts to TCE.

4.4 Calling Convention

TCE does not specify a calling convention scheme that the compiler has to follow. The compiled programs are fully linked and there are no system calls or external libraries. Therefore the compiler can use any calling convention, which can be changed without any binary compatibility issues.

The current implementation uses a calling convention which is outlined in the following rules.

- All parameters are stored in the stack by the caller. Parameters of type i1, i8 and i16 are always promoted to 32 bits to avoid problems with variadic function calls.
- All general purpose registers are caller-saved, i.e., the caller is responsible for saving all registers it has live values into the stack before a call.
- The return address of a call is stored in the return address register by hardware when a call instruction is invoked. The return address is saved to the beginning of the stack frame in function prologue.
- Function return value is stored in the return value register by the function epilogue. 64-bit return values are split into two 32-bit parts, one of which

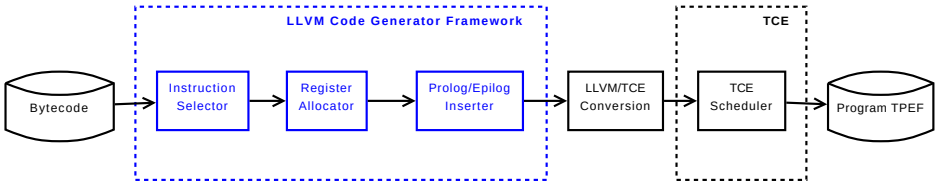

Figure 4.2: TCE Backend.

is returned in a secondary return value register. Returning of values that do not fit in the return value registers is handled by the LLVM code generator framework.

- Callee is responsible for restoring the stack pointer and return address registers to the state they had on function entry.

4.5 TCE Backend

One of the main design goals of the TCE backend is to reuse as much code from the LLVM code generator framework as possible. However, modeling a customizable architecture template as a target machine and the use of an external scheduler set limitations requiring a design that differs from the standard LLVM code generation organization.

TCE code generation can be considered to consist of two main phases. First, the LLVM code generator framework is utilized to produce a sequential target machine program. The sequential program is then scheduled to a parallel program using a TCE instruction scheduler. The main code generation passes are illustrated in Figure 4.2.

The instruction selector, register allocator and prologue/epilogue code inserter passes are based on the LLVM code generator framework. These passes produce a sequential program where all instructions are selected as target machine instructions and all registers are allocated to physical registers.

The sequential program is represented as an LLVM data structure, which must be converted to a TCE program object model for the scheduler. The conversion is handled by *LLVMPOMBuilder*, which is implemented as a separate LLVM code generator pass. The sequential POM can then be scheduled and a parallel TPEF written for simulation and binary program image generation. The TCE instruction scheduler [23] is implemented as an independent library and not covered in this thesis.

The main design problem of the code generator is the requirement for retargetability. The LLVM part of the code generator only generates a sequential program which simplifies the adaptation to the target machine. The scheduler is responsible for allocating most of the physical resources while taking care of low level limitations

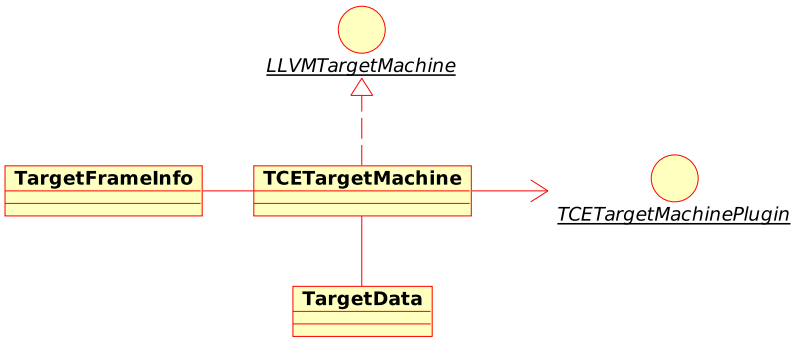

Figure 4.3: TCE Target Machine.

of the architecture such as operation latencies. This means that the LLVM part of the code generator is only required to retarget itself at an abstract level to the instruction and register sets of the target machine. The rest of the LLVM target machine properties, such as type sizes and calling conventions are static across the architecture template.

The dynamic retargeting of the backend is achieved by implementing the TCE target machine as a wrapper, which loads the target instruction and register set models from a target machine specific plugin. The plugins are generated at compile runtime from a machine object model of the target architecture.

4.6 TCE Target Machine

The TCE backend implements an LLVMTargetMachine that models the templated TCE architecture. The target machine implementation is divided to a static part which models the hardcoded properties of the architecture template, and a dynamic part which models the customizable properties of concrete TCE machines. Only the static part of the target machine is compiled to the TCE backend library. The machine specific dynamic part is loaded from a plugin, which can be compiled independently for each target machine without recompiling the whole backend library. High level design of the TCE target machine is illustrated in Figure 4.3.

TCETargetMachine implements the *LLVMTargetMachine* interface which is used by the code generator to access the target architecture model and to create an instruction selector pass. Only the data memory layout and frame information of the machine model are implemented in the static backend and returned directly. The customizable part of the architecture model is contained in machine specific plugins which implement the *TCETargetMachinePlugin* interface. The instruction selector pass is generated from the instruction set descriptors of the target machine and therefore it is also implemented in the target machine plugin.

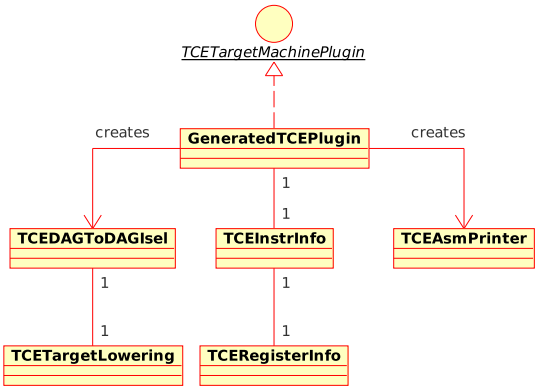

Figure 4.4: Target Machine Plugin.

TargetData defines the data memory layout of the TCE architecture template. The layout is hardcoded and does not change between individual machines. The memory layout is defined to have the following properties:

- Pointer size is 32 bits,
- Data memory is big-endian,
- LLVM i1 and i8 types have 8 bit size and alignment in memory,
- LLVM i32 and f32 types have 32 bit size and alignment in memory, and
- LLVM i64 and f64 types have 64 bit size and 32 bit alignment in memory.

TargetFrameInfo is also equal to all machines. It defines that the stack grows down, stack contents are aligned to 32 bits on entry to functions, and the offset to local area of a stack frame is -4 bytes.

4.7 Target Machine Plugin

When the TCE backend library is instantiated, a machine object model of the target machine is passed to the backend constructor. The backend generates source code for a plugin containing the part of the target machine model which varies between individual machines. The plugin is then compiled and loaded to complete the target machine model. Figure 4.4 shows a class diagram of the generated plugin.

GeneratedTCEPlugin is the main class of the plugin. It implements the *TCE-TargetMachinePlugin* interface, which is used by the *TCETargetMachine* class to access the plugin.

TCEInstrInfo derived from the LLVM *TargetInstrInfo* base class models the target machine instruction set and implements various code generation methods of the *TargetInstrInfo* interface. The code generation methods do not change between target machines and are therefore static source code for the plugin. The implemented virtual code generation methods of the base class are:

- *copyRegToReg:* Generates code for copying values between registers by inserting move instructions,
- *loadRegFromStackSlot:* Inserts a load instruction for loading value in stack to a register,
- *storeRegToStackSlot:* Inserts a store instruction for storing value in a register to stack, and
- *InsertBranch:* Inserts unconditional branch instructions.

The code generated by these methods only use instructions that are part of the minimal required opset. The same code can therefore be generated for all target machines.

The class also has methods for querying properties of target instructions, such as if an instruction is a move between registers, or a load or store to stack. These methods are related to code that is generated by the static code generation methods using the minimal operation set, and do not require machine specific implementation.

The instruction set model consists of instruction descriptions and enumerations generated from target descriptor files of the target machine by the *TableGen* code generator tool, described in Section 3.6.

TCERegisterInfo class models the target machine register set and contains various code generation methods related to stack frame handling. The class is derived from the LLVM *TargetRegisterInfo* base class. The following virtual code generation methods of the base class are implemented.

- *eliminateFrameIndex* generates code to replace abstract frame index operands with code that calculates absolute addresses of items in stack from the stack pointer register and an offset operand.
- *eliminateCallFramePseudoInstr* replaces pseudo instructions adjusting stack frame size with code that adjusts the stack pointer register value.
- *emitPrologue* is used by the prologue and epilogue code insertion pass to generate function prologue code. The generated code initializes function stack frames by saving return address register value to the stack and reserving space for local variables.

- *emitEpilogue* is used by the prologue and epilogue code insertion pass to generate function epilogue code. The generated code restores the saved return address register value from stack, frees space reserved by the stack frame and returns from the function call.

These functions generate code using only instructions that are part of the minimal required opset. The generated code is therefore same for all target machines and the functions have an implementation which is static source code for the plugin.

The register set model consists of register info descriptions generated from *target descriptor* files of the target machine by the *TableGen* tool.

TCEDAGToDAGISel implements an LLVM DAG to DAG instruction selector for the target machine. The derived class implements selection methods which convert SelectionDAG nodes to target instruction nodes. Most of the nodes can be selected with instruction selection methods which are generated from the target description files of the target machine. The methods are generated from the instruction patterns which are part of the target instruction descriptors. The SelectionDAG nodes that are not selected by the *TCEDAGToDAGIsel* methods are lowered to target instructions with *TCETargetLowering* phase. The following instruction and operand nodes are selected to temporary target nodes which are lowered to target instructions later.

- Conditional branch instructions are selected to pseudo instructions, which are converted to guarded jumps in the *LLVMPOMBuilder* pass,
- Unconditional branch instructions are selected to pseudo instructions, which are converted to jumps in the *LLVMPOMBuilder* pass,
- Frame index operands are selected to target frame index operands, which are lowered by the *eliminateFrameIndex* method of the *TCERegisterInfo* class, and
- Memory address operands are selected to target constants and address operands which are lowered in later phases of code generation.

TCETargetLowering is an implementation of *LLVMTargetLowering* base class. Target lowering is the second phase of instruction selection, which lowers *SelectionDAG* nodes that were not selected by *DAGToDAGISel* phase. The class constructor also initializes available register types and which nodes are expanded and promoted by the code generator framework. The following lowerings are initialized in the constructor.

- General purpose register classes for i1, i32 and f32 types are defined, allowing LLVM to use operands of these types in the target code. Operands of other types are lowered to instructions using these types.
- Various instructions are expanded and promoted to instructions that are easier to lower.
- Integer division, modulo, rotate and multiplication instructions are set to be expanded to emulation function calls if the target machine does not have the corresponding instructions available.

The class implements the following functions to lower instructions that require TCE specific custom lowering.

- *LowerCallTo* lowers function call instructions. First, the stack space required by the function call arguments is calculated. The calculated space is set as argument size for a *CALLSEQ_START* pseudo instruction node which is inserted to mark the start of a call sequence. Next, a sequence of stores are inserted to store the call arguments to stack. Arguments of types i1, i8 and i16 are extended to 32 bits. The actual target call instruction is inserted after the stores. The call instruction is followed by a move which copies the return value from the return value register. Finally, a *CALLSEQ_END* node with the argument size operand is inserted to mark the end of the call sequence.
- *LowerArguments* lowers *FORMAL_ARGUMENTS* nodes which have the incoming arguments of functions as operands. The node is lowered to a sequence of loads which loads the function arguments from the stack. Arguments of type i1, i8 and i16 are truncated to their original bit width from the extended 32-bit value passed in the stack.
- *LowerRET* lowers LLVM return instructions in functions having a return value. The instruction is lowered to a move which copies the return value to the return value register followed by a target return instruction. Return instructions in functions that do not have a return value are instruction selected in the *TCEDAGToDAGISel* phase.
- *LowerSelect* lowers *SELECT* instructions, which select result between true and false value operands based on a boolean condition operand. The instruction is lowered to a pseudo instruction node which is converted to guarded moves in the *LLVMPOMBuilder* pass.
- *LowerOperation* lowers global addresses and function constant pool entries to target nodes that are handled by the *LLVMPOMBuilder* pass.

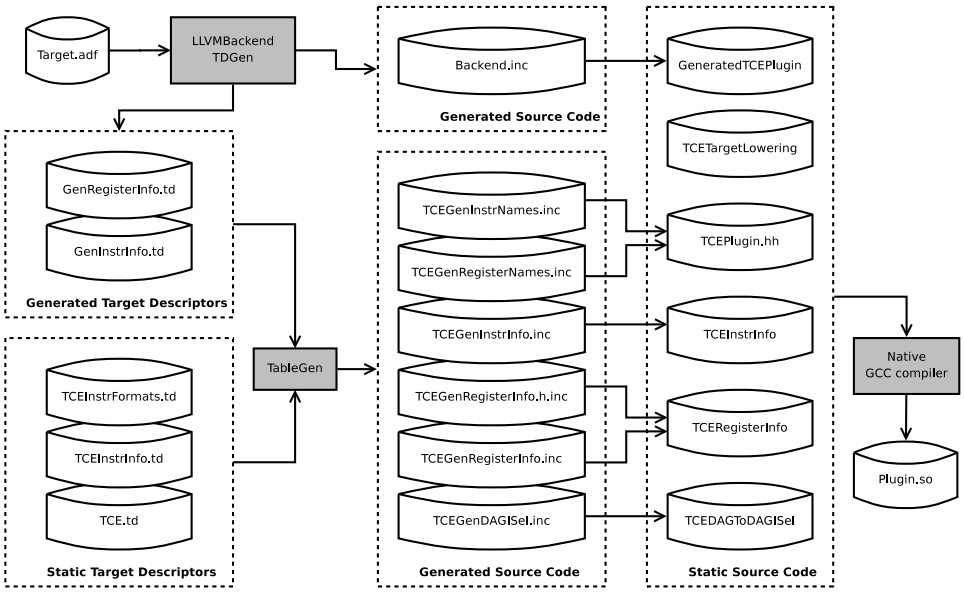

Figure 4.5: Target Machine Plugin Generation.

4.8 Plugin Generation

The target machine plugins are generated by the backend when a new backend is instantiated. Source code for a target machine plugin consists of static skeletons for each LLVM target machine class. The skeletons contain all code generation methods that can have the same implementation for each machine. The rest of the methods are generated by the *TDGen* code generator which generates target descriptor files of the target machine instruction and register sets. The descriptors files are processed with the LLVM *TableGen* tool, which generates implementations for various code generator callback functions and data structures. The code generated by *TableGen* is added to the skeletons as preprocessor includes. The plugin generation phases and intermediate files are illustrated in the Figure 4.5.

TDGen generates two target descriptor files: *GenInstrInfo.td*, which contains the instruction set descriptors and *GenRegisterInfo.td* which contains the register descriptors. The generated files are processed by *TableGen* along with three static .td files: *TCEInstrFormats.td* contains the TCE instruction class records, *TCEInstrInfo* contains instruction records required by all target machines and a top level *TCE.td* which includes all descriptor files and required LLVM header files to one entity. In addition to the two .td files, *TDGen* generates source code for various helper methods in *Backend.inc*.

The generated source code files are included in the class skeletons and compiled with a native compiler to a plugin, which can be loaded by the *TCETargetMachine*.

4.8.1 Register Set Descriptors

The whole register set description consists of one autogenerated *GenRegisterInfo.td* file. Seven general purpose register classes are always defined: *Ri1*, *Ri8*, *Ri16*, *Ri32* and *Ri64* for integers, and *Rf32* and *Rf64* for floating point values. The register classes are divided to integer and floating point register types because LLVM does not currently support declaring one register class for both integer and floating point values of equal width. Only the *Ri1*, *Ri32* and *Rf32* register classes are added in *TCETargetLowering* as register classes that are available for the code generator, rest of the classes are currently unused. All TCE register classes are derived from the *TCEReg* base class, which defines classes as a member of the *TCE* name space, and allows passing a list of register aliases.

The register set description is generated by first analyzing the register set of the target machine. *TDGen* iterates through all register files and registers in the target machine architecture model. Registers are added to register classes of corresponding width with the following exceptions.

- If a register has a guard on a target machine bus, it is added as a member of the *Ri1* register class so it can be used as a condition for conditional moves. This is done even if the register file is wider than one bit.
- If the machine is not fully connected, the last register of each register file is reserved for scheduler to route values between ports with missing connection.

The generated register classes have the following registers reserved for special use:

- One 32-bit register is reserved to be the stack pointer register.
- One 32-bit register is reserved to be the return value register.
- One 32-bit register is reserved to be always available for the code generator. It is used for calculating absolute address values from memory operands with base address and an offset operand. The register is also used when half of a 64-bit return value has to be returned in an additional register with the return value register.
- One 64-bit register is reserved for 64-bit return values if a 64-bit register file exists.

In addition to the general purpose register classes, a special register class is defined containing one register for the return address port of the target machine control unit. The register records do not have any TCE specific attributes. The records are enumerated, and an external helper function is generated in the *Backend.inc*

file, which maps the register enumerations to target machine register file names and register indices.

4.8.2 Instruction Set Descriptors

The instruction set is defined in three target descriptor files. *TCEInstrFormats.td* and *TCEInstrInfo.td* contain the static descriptors, which stay the same for each target machine. *TCEInstrFormats.td* contains the instruction format class template of TCE instructions. *TCEInstrInfo.td* contains descriptors for instructions and operand types that are always required by the code generator. The two static files are installed as source code files which are used when a backend plugin is compiled. The third file, *GenInstrInfo.td* which contains the target machine specific instruction descriptors is generated at compiler runtime by the *TDGen* code generator of the TCE backend library.

Instruction format. Since most of the instruction descriptors are automatically generated, there is no need to utilize any complex instruction format structures which reduce repetition of manually written patterns. A flat hierarchy of descriptors is easier to generate automatically. Only one simple instruction format class *InstTCE* is defined and used by all instructions. The class is defined in the static *TCEInstrFormats.td* file to have the following structure:

```
class InstTCE<dag outOps, dag inOps, string asmstr,
              list<dag> pattern> : Instruction {

   let Namespace = "TCE";
   dag InOperandList = inOps;
   dag OutOperandList = outOps;
   let AsmString = asmstr;
   let Pattern = pattern;
}
```

The *InstTCE* instruction format has the usual input and output operand lists and an instruction selection pattern. The assembly string is optional and only used for debugging purposes because an assembler is not used. The instruction format does not contain any TCE specific attributes. The mapping of instruction enumerations to the OSAL instructions is handled by an external helper function, which is generated in the *TCEBackend.inc* file.

The operation pattern is also optional. It is required by the instruction selector to be able to automatically exploit the instruction, but can be omitted if automatic instruction selection is not needed for the instruction. For example, an instruction

which is only inserted by the code generation methods of *TCETargetLowering* does not require a pattern.

The *Instruction* base class attributes of an instruction record that are not in the instruction class template are defined by enclosing the instruction record in a *let* block. The following example sets a conditional branch instruction record to have the branch and terminator flags set to true:

```
let isTerminator = 1, isBranch = 1 in {
    def TCEBRCOND : InstTCE<(outs), (ins I1Regs:$gr, i32imm:$dst),
        "? $gr $dst -> jump.1;", []>;
}
```

Static instruction descriptors. *TCEInstrInfo.td* contains instruction records which are always required for code generation, but cannot be easily generated automatically. The instruction records consist mainly of pseudo instructions that do not have corresponding instructions in a target machine function unit. The pseudo instructions are converted to target machine moves in the *LLVMPOMBuilder* pass. The pseudo instructions include the following records:

- Move instruction records for all legal combinations of source and destination operand types,
- Various control flow instructions of the target machine control unit,
- Pseudo instructions which adjust stack frame size, and
- *SELECT* pseudo instructions which select one of two operands conditionally, based on a condition operand.

In addition to the pseudo instruction records, the file includes patterns which select load and store instructions for operands that require extension or type conversion. The static descriptors also include patterns for memory address operands. The memory address operands are selected to target machine nodes, which are converted to absolute addresses in the *LLVMPOMBuilder* pass after the program data memory has been laid out.

Generated instruction descriptors. *GenInstrInfo.td* contains the target machine specific instruction descriptors which are generated by the *TDGen* module. The machine specific instruction records consist of records for all target machine instructions which have known semantics. The file also contains emulation patterns for instructions that are required, but not supported by the target machine.

The instruction descriptor generation is done in the following main steps.

1. The function units of the target machine are examined to create a list of supported abstraction layer operations.
2. A list of required operations is initialized with a hard-coded set of abstraction layer operation names.
3. Instruction records with all combinations of legal operand types are generated for each target machine operation having known semantics. The operation is removed from the list of required operations initialized in the previous step if it is on the list.
4. The remaining operations in the required operation list are the missing operations which must be emulated. An emulation pattern is generated for each missing operation.
5. A special instruction record is generated for the *CALL* instruction of the target machine control unit.

Instruction records of the target machine operations are generated from OSAL operation descriptions. The first step is to check if the operation has known semantics. *TDGen* contains a hard-coded list of primitive OSAL operations which have known semantics mapped to LLVM instruction patterns. In order to generate an instruction record, the OSAL operation must be part of the hard-coded primitive set, or the operation must have a data flow graph consisting of known primitive operations. Operations with unknown semantics are skipped and cannot be used automatically by the compiler. These *custom operations* can only be used explicitly in the compiled program source code by invoking them with preprocessor macros as will be described in Section 4.10.

The second step is to determine legal combinations of operand types. OSAL does not define bit-width for integer operands, so additional logic is required to determine the combinations of operand types that can be used in the instruction records. Instruction patterns with an output operand in *Ri32* register and input operands in combinations of *Ri32* registers and 32-bit immediates are always generated for operations with integer operands. Instruction records with operands in *Ri1* registers and as 1-bit immediate values are restricted to a hard-coded set of operations, which are known to work with 1-bit input and output operands.

The third and final step is to generate an instruction selection pattern and write an instruction record for all combinations of legal operand types. The primitive OSAL operations are mapped to corresponding LLVM instruction pattern templates. The templates are hard-coded strings which can be formatted by placing operand strings in the template. The template strings can also be nested to generate instruction selection patterns from OSAL operation semantics DFGs.

A simple example is the OSAL *ADD* operation, which is one of the primitive operations with known semantics. The corresponding instruction pattern template is

```
"add %1%, %2%"
```

where *%1%* and *%2%* mark the positions for the input operand strings. One of the legal operand combinations for *ADD* has one input in *Ri32* register and one as a 32-bit immediate. The instruction pattern template is formatted with strings corresponding to the operand types, resulting in the following pattern fragment:

```
"add I32Regs:$op1, (i32 imm:$op2)"
```

The result of the add operation is written in a *Ri32* register, which has to be defined in the instruction selection pattern. This is achieved by wrapping the generated pattern fragment in a pattern which sets the result in the correct type of operand. The complete instruction selection pattern for *ADD* with the chosen types of operands is:

```
"(set I32Regs:$op3, (add I32Regs:$op1, (i32 imm:$op2)))"
```

The instruction record input and output operand lists are generated according to the operand types. The assembly string is not required and it is currently left empty for all generated instruction records. Now the whole instruction record can be written, taking the following form:

```
def ADDri : InstTCE<
    (outs I32Regs:$op3), (ins I32Regs:$op1,i32imm:$op2), "",
    [(set I32Regs:$op3, (add I32Regs:$op1, (i32 imm:$op2)))]>;
```

The instruction record name *ADDri* is generated by combining the operation name with a letter for each input operand type. The generated name is added to a helper function in *Backend.inc* file, which maps the instruction records back to OSAL operation names. The helper function is used by the *LLVMPOMBuilder* pass to invoke correct OSAL operations when building a *POM*.

The same process is repeated for all operations with known semantics having any combination of legal operand types, aiming to generate as effective instruction selector from the instruction records as possible.

Emulation patterns. An instruction selection pattern must be generated for all OSAL operations that are required, but not supported by the target machine. The emulation patterns are generated as selection DAG patterns which match the LLVM

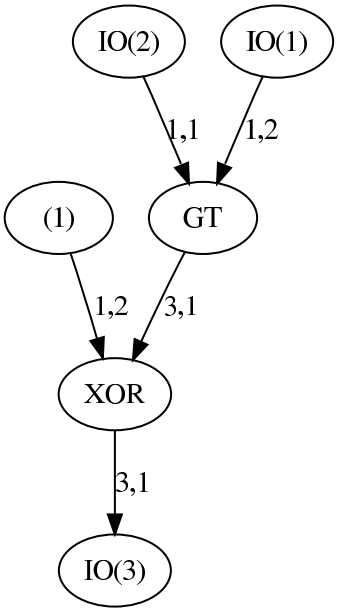

Figure 4.6: An emulation DFG for the *GE* operation.

instruction corresponding to the missing operation to a combination supported operations.

The generation of an emulation pattern is based on the OSAL semantics DFGs of the missing operation. In order to write an emulation pattern, the missing operation must have a DFG which defines the semantics of the operation with operations that are supported by the target machine.

An emulation pattern is generated by first requesting an emulation DFG from the *OperationDAGSelector* module. A list of supported operations and the name of the missing operation are passed to the module. The module returns the semantics DFG with the least number of operations that are supported by the target machine.

As an example, let us assume that a target machine is missing the required greater-or-equal *GE* operation. A list of supported operations, including the supported exclusive-or *XOR* and greater-than *GT* operations are passed to the *OperationDAGSelector*, which returns a DFG illustrated in Figure 4.6 (note the reversed operands for *GT*).

Emulation patterns for all required operand type combinations are generated from this DFG. The patterns to match the missing operation are generated using the same method which is used when generating instruction patterns of supported operations. The pattern of resulting instructions is generated from the DFG in a similar manner, with the exception of using the generated names of the supported operations instead of LLVM instruction names.

One of the resulting emulation patterns, having both input operands in *Ri32* class registers has the following final form:

```
def : Pat<(setge I32Regs:$op1, I32Regs:$op2),
          (XORrr (GTrr I32Regs:$op2, I32Regs:$op1), 1)>;
```

A similar selection DAG pattern is generated for all required combinations of operand types.

***CALL* Instruction.** The calling convention described in Section 4.4 states that all general purpose registers are caller-saved. The *Instruction* base class for instruction records has a *Defs* attribute, which is a list of registers that the instruction may modify. The register saving is achieved by enclosing all *CALL* instruction records in a *let* block, which sets the *Defs* list to contain all general purpose registers. This forces the code generator to save any live values in GPRs before invoking a call, and to restore the values after the call. The register list is target machine dependent and must be generated dynamically for each target machine.

4.9 LLVMPOMBuilder

LLVMPOMBuilder is a pass which converts LLVM target dependent representation of a program to a TCE Program Object Model for scheduling. The pass is implemented as an LLVM *MachineFunctionPass*. LLVM *MachineFunctionPasses* are code generator passes that are executed on each function of the target dependent representation of the compiled program. The passes are executed in three stages. First the pass is initialized with a *doInitialization()* method. Next, *runOnMachineFunction()* is called for each function of the program module. Finally, the pass execution is finalized with the *doFinalization()* method.

The conversion process consists of laying out the program global variables in the data memory, and converting the program instructions to a POM function by function. The converted representation is already using target machine instructions and registers, so individual instructions and operands can be converted in a straightforward manner. The only problem building the POM are the references between the program data, functions, basic blocks, instructions and operands. For example, a global variable in the data memory might be initialized with a function pointer, or an instruction might have a basic block label as an operand. For this reason, the program is built using placeholders for the elements that can not be initialized until the whole program is built. The placeholders are fixed to reference to correct entities when the conversion is being finalized and all symbol locations are resolved.

The conversion is done in the following order:

1. Global initialized data is laid out to data memory. Absolute location addresses are added to bookkeeping.
2. Global uninitialized data is laid out to data memory and locations are added to bookkeeping.
3. The program instructions are converted function by function to POM representation. Instruction addresses of first instructions in functions and basic blocks are added to bookkeeping.
4. POM data definitions are built for initialized and uninitialized global data.
5. An *end* symbol is generated at the end of the reserved data memory area.
6. References to the *end* symbol are fixed.
7. References to basic blocks are fixed according to bookkeeping.
8. References to code labels are fixed according to bookkeeping.
9. Stack pointer initialization code is inserted to the beginning of the program.

Phases 1 and 2 are done in the *doInitialization()* method. Phase 3 is done by the *runOnMachineFunction()* method. Phases 4-9 are done in the *doFinalization()* method.

4.10 Operation Macros

An assembler is not part of the TCE compiler toolchain, and therefore inline-assembly is not supported. However, low level programming is often required, especially when a programmer wants to use complex custom operations that cannot be exploited automatically by the instruction selector. TCE circumvents the lack of inline assembler by defining preprocessor macros, which allow hardware operations to be invoked directly in C source code.

The operation macros can be used by including the *tceops.h* header file. The header is automatically generated during compilation and contains macros for all target machine operations. The syntax to invoke an operation is

```
_TCE_opname(input operand 1, ..., output operand 1, ...);
```

For example, some TCE test suite target machines contain an address generator operation "AG" with four input operands *i1, i2, i3, i4* and two output operands *o1, o2*. The operation can be invoked with the following syntax:

```
_TCE_AG(i1, i2, i3, i4, o1, o2);
```

The operation macros are defined as dummy inline-assembly blocks, where the assembly string contains only the operation name. The macro parameters are set as input and output variables for the inline assembly block according to OSAL description of the operation. The inline assembly is passed as an LLVM inline assembly instruction to the *LLVMPOMBuilder* pass where it is converted to a sequence of moves invoking the operation corresponding to the name in the assembly string.

5. VERIFICATION AND BENCHMARKING

The compiler was verified and benchmarked using the Embedded Microprocessor Benchmark Consortium (EEMBC) Digital Entertainment Benchmark (DENBench) software suite [24]. The benchmark suite consists mainly of video, audio, and image decoding and encoding benchmarks written in the C programming language.

The benchmarks were run with several TTA machines, starting with a near-minimal architecture configuration. The architecture was then expanded by adding more registers and operations incrementally, to verify correct compilation and to benchmark the ability of the compiler to exploit additional registers and operations.

5.1 Testing Setup

The testing was performed using a development version of the TCE toolchain with a llvm-gcc-4.2.1 -based compiler frontend and a LLVM-2.5 -based compiler backend. The compiled programs were simulated using the compiled simulation engine of the TCE architecture simulator.

The DENBench test programs produce output that can be verified for correct compilation with two types of verification data. The programs that only use integer arithmetic produce an easily verifiable checksum, but the tests that use floating point arithmetic have results that depend on the floating point presentation and precision of the target machine. The floating point tests were therefore verified by calculating the peak signal-to-noise ratio (PSNR) of the output data compared to the original data that was encoded and decoded. The results were compared to the results of a native GCC-compilation.

A subset of the DENBench benchmark suite was chosen for benchmarking. The whole benchmark contains multiple datasets for each test program, but due to the long simulation and scheduling times, only one dataset of each test case was chosen for the final benchmarking.

5.1.1 Test Machine Architectures

The first test machine architecture *8reg.adf* contained the minimal operation set required by the compiler to compile arbitrary C-programs. The number of registers was nearly minimum, divided to two register files, one having eight 32-bit registers and the other two 1-bit registers having transport bus guards to support conditional

execution.

The first test machine is expanded to *16reg.adf*, *24reg.adf* and *32reg.adf* by adding one additional register file of each type on each increment. Consequently, the last register set expansion test machine has 32 32-bit registers divided to four register files and eight 1-bit registers divided to four register files.

The *32reg.adf* was then expanded by incrementally adding operations and function units to the previous test machine configuration. The fifth test machine *alu.adf* adds integer operations, such as half- and quarter-word sign-extension integer comparison operations. The sixth test machine *mul.adf* adds a function unit with *MUL* integer multiplication operation. The seventh test machine *div.adf* adds a function unit with signed and unsigned integer division and modulo operations. The eight and final test machine configuration *fpu.adf* adds a single-precision floating-point function unit for hardware floating-point arithmetic support.

The automatic utilization of custom operations was tested as a separate test case with a *mac.adf* architecture. The architecture was based on the *div.adf* architecture, with an addition of a *MAC* function unit. The function unit contains a custom multiply-accumulate (MAC) operation, which computes the product of two integer operands summed to a third integer operand.

5.1.2 Test Cases

The verified and benchmarked DENBench test suite is divided to the following mini-suites:

MPEG The MPEG algorithm mini-suite includes a MP3 audio decoder, a MPEG-2 video encoder and decoder and a MPEG-4 video encoder and decoder. The encoding and decoding algorithms use fixed-point integer functions, except for a separate MPEG-2 video encoding algorithm, which uses single-precision floating-point encoding.

Cryptography The cryptography mini-suite includes the Advanced Encryption Standard (AES) algorithm for public-key cryptography, and a Huffman data decompression test.

Digital Image Processing The digital image processing mini suite includes color space conversion tests and JPEG image compress and decompress tests. The color space conversions tested are Red-Green-Blue to Cyan-Magenta-Yellow-Key (RGB-CMYK), Red-Green-Blue to luminance-chrominance (RGB-YIQ), and Red-Green-Blue high-pass grey-scale filter (RGB-HPG).

Test Case	8reg.adf	16reg.adf	24reg.adf	32reg.adf
AES Encryption	6,264,499,994	4,466,047,052	4,486,888,618	4,281,280 926
Huffman decoder	17,171,346	13,237,365	12,074,356	11,999,862
MP3 Decoder	80,917,223,868	52,849,307,890	53,404,066,109	52,226,334 542
MPEG2 Fixed Point Decoder	9,546,900,307	6,918,972,000	6,324,530,344	6,303,739,676
MPEG2 Fixed Point Encoder	31,721,047,907	14,995,595,434	12,837,263,273	12,420,274,298
MPEG2 Floating Point Decoder	9,484,689,575	6,878,763,327	6,284,616,128	6,263,998,257
MPEG2 Floating Point Encoder	92,748,324,214	59,332,178,569	56,909,735 974	56,126,620,307
MPEG4 Decoder	7,899,586,664	5,840,422,082	5,259,561,810	5,260,183 293
MPEG4 Encoder	9,671,106,214	5,096,814,413	4,479,466,232	4,384,969 549
JPEG Decompress	70,237,028	47,393,649	46,715,535	46,430,077
JPEG Compress	92,048,335	57,095,334	54,867,017	53,941,416
RGB-CMYK Conversion	81,920,332	63,978,956	61,688,361	61,934,927
RGB-HPG Conversion	42,399,960	28,952,858	28,723,261	28,729,041
RGB-YIQ Conversion	144,035,421	102,770,795	101,462,594	101,080,386

Table 5.1: **Total cycle counts while extending the register set.**

5.2 Results

The total cycle count results of register set extension test machines are shown in Table 5.1, and the total cycle count results of operation set extension test machines are shown in Table 5.2. All tests were run with full compiler optimizations (-O3).

The combined benchmark results of the register and operation set extension are visualized in Figure 5.1. The cycle counts of each test are normalized to a percentage of the cycle count of the first *8reg.adf* test machine to give a comparative view of the effects of expanding the processor architecture with different test cases.

The register set extension results of the *8reg.adf* to *32reg.adf* show that the compiler was capable of utilizing the additional registers. However, the registers were added in large increments and only the first step shows a major benefit in the cycle counts, and only some test cases had decreased cycle counts with high number of registers. This was an expected result since spilling of register values to the memory with the tested programs becomes negligible after the addition of the second register file of each type. With additional register files the compilation becomes constrained by other factors limiting the program parallelization.

Some tests had even a slightly increased cycle count with additional registers. This is due to the sensitivity of the TCE scheduling algorithm to the input program. Even a minor modification to the input program can affect the schedule greatly, sometimes worsening the results slightly.

Test Case	alu.adf	mul.adf	div.adf	fpu.adf
AES Encryption	4,542,904,024	4,540,030,984	1,160,345,306	1,160,345 255
Huffman Decoder	11,642,552	8,825,467	9,052,383	8,985,644
MP3 Decoder	48,520,367,083	9,059,358,249	1,479,281,602	1,467,197,925
MPEG2 Fixed Point Decoder	6,137,917,674	5,084,777,212	5,085,924,323	5,066,232,403
MPEG2 Fixed Point Encoder	11,911,073,035	5,836,516,113	5,054,462,375	4,938,783,751
MPEG2 Floating Point Decoder	6,101,594,728	5,080,679,371	5,081,818,369	5,062,126,469
MPEG2 Floating Point Encoder	52,457,356,242	24,456,512,313	23,701,893 525	5,245,135,767
MPEG4 Decoder	5,111,721,660	4,766,445,482	4,762,085,052	4,752,243 766
MPEG4 Encoder	4,183,065,148	1,446,986,368	1,439,220,215	1,439,114 164
JPEG Compress	44,264,615	17,368,372	17,464,110	17,467,953
JPEG Decompress	50,905,910	16,753,591	15,646,123	15,646,056
RGB-CMYK Conversion	58,911,880	48,194,394	48,424,460	48,147,588
RGB-HPG Conversion	26,920,175	9,715,088	9,637,959	9,637,905
RGB-YIQ Conversion	94,255,793	24,339,522	24,339,195	24,108,743

Table 5.2: **Total cycle counts while extending the operation set.**

The first operation set extension step with the *alu.adf* test machine added operations which can be emulated with simple patterns of operations existing in the previous test machine. The compiler was able to utilize the new operations, but most test cases showed only small improvements in the cycle counts.

The second operation set extension test machine added an integer multiplication operation. The operation is used heavily by most of the test programs and the emulation of it is slow. As expected, the effect of the compiler utilizing the operation can be clearly seen from the diagram. For example, the MP3 test case had a cycle count decreasing more than 80% compared to the previous test machine.

The *div.adf* added a divisor function unit to the *mul.adf* architecture. The division operation is slow to emulate, but only the AES and MP3 test programs rely on it in their core algorithm loops. However, the results show that the compiler utilized the divisor function unit operations and the two test cases relying on the operations had major decreases in the cycle counts.

The final *fpu.adf* added floating point operations. The MPEG-2 floating point encoder was the only test case that utilizes floating point values in the benchmarked algorithm and consequently it was the only test case which had clearly decreased cycle count. However, the test case proved that the compiler is capable of utilizing floating point operations.

Utilization of the MAC operation in the *mac.adf* architecture was tested as a

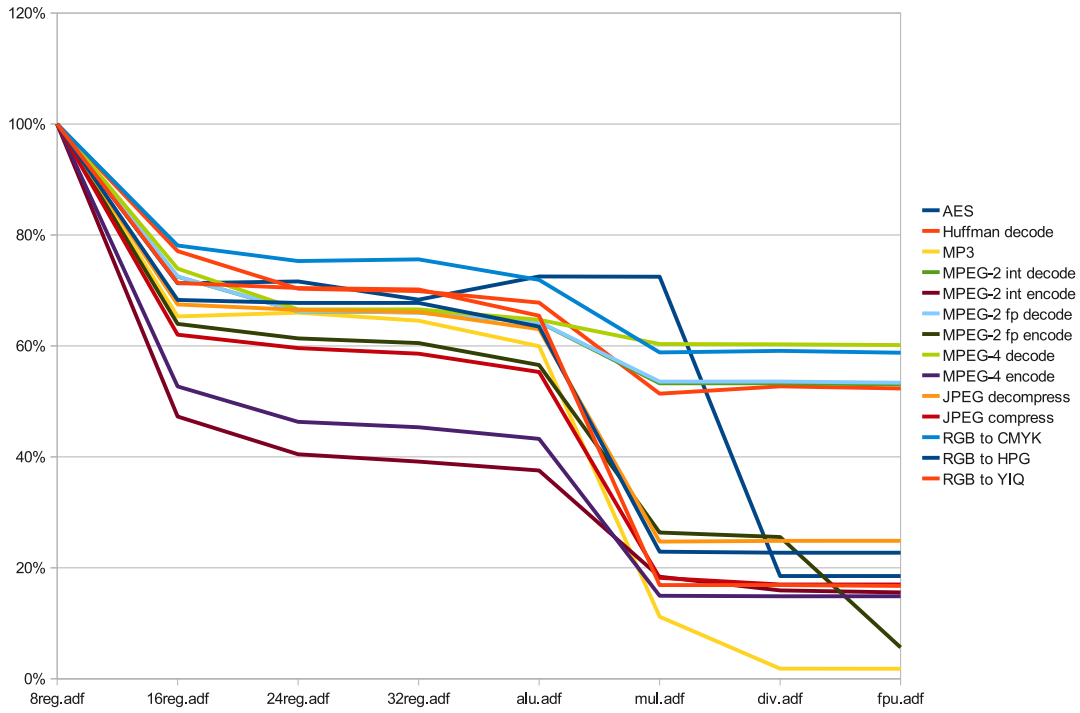

Figure 5.1: Benchmark results normalized to the first test machine.

separate test case with a different version of the TCE toolchain. The results are therefore only compared with the *div.adf* results with the same toolchain version. The highest utilization of the operation was achieved with the MP3 decoder benchmark, where the instruction pattern matched operations in the main decoding loop. Results for other test cases are omitted.

To analyze the effects of the addition of the MAC operation, the number of executions of each operation in the *mac.adf* was compared with the corresponding results with the *div.adf* architecture. As expected, only the MUL and ADD operations had significant differences in the execution counts. The number of these operations and the total number of all executed operations are presented in the Table 5.3.

The comparison of the results shows that approximately 192 million multiply accumulations were performed using the MAC operation. However, the total number of operations executed decreased by less than 140 million operations, because part of the matched multiply-accumulate patterns had an intermediate multiplication result which was used as an input operand for more than one ADD operation.

The benefit of having the MAC operation was further diminished by the lesser degree of parallelization achieved by the instruction scheduler. The program execution time decreased by approximately 2.1 %, from 1,576,173,008 to 1,543,460,040 clock cycles.

The purpose of this test was to demonstrate the ability of the compiler to auto-

	div.adf	**mac.adf**	difference
MUL	268,469,874	133,805,319	-134,664,555
ADD	841,446,719	646,930,374	-194,516,345
MAC	-	191,858,218	+191,858,218
Other operations	1,297,327,995	1,294,872,357	-2,455,638
Total	2,407,244,588	2,267,466,268	-139,778,320

Table 5.3: **Operation execution counts for the MP3 decoder test case with and without the MAC operation.**

matically utilize operations defined with custom operation patterns. The results of the MP3 decoder test case show that the compiler backend can achieve relatively high utilization of a simple operation pattern in realistic programs. In general, the benchmark results show that the compiler is capable of adapting itself to architectures with varying operation and register sets, and it can exploit additional resources of the target machines.

6. CONCLUSIONS

This thesis introduced a new compiler for the TCE toolchain, excluding the instruction scheduler which is designed and documented separately. The compiler was implemented as a backend for the LLVM compiler infrastructure.

The thesis gave a brief introduction to the TTA concept and described the TCE codesign toolchain for designing and programming TTA processors. The description of the role of the compiler in the toolset was followed by a description of the relevant compiler concepts and the introduction of the LLVM compiler infrastructure.

The main part of this thesis is the documentation of the implementation details of the TCE-LLVM compiler backend. The backend differs from most compiler backends by the compilation target machine, which is a processor architecture template instead of a static processor design. The backend was required to adapt to different TTA architectures derived from the TCE architecture template, which introduced complex design problems having to be solved.

In addition to the retargetability, the requirements for the compiler were correctness of compiled programs and the requirement of providing a reliable and extensible base for further compiler development. The compiler design also aimed to provide powerful modern compiler optimizations, which was achieved by choosing the LLVM as the basis for the compiler.

The compiler was verified with various test cases and benchmarks. The results for the most important benchmark, the *EEMBC DENBench Digital Entertainment Benchmark Suite* are presented in this thesis. All test results were successfully verified and produced either the correct checksum, or PSNR result similar to the native benchmark results. The results also confirmed that the compiler was able to adapt to different templated architectures and was able to utilize additional resources in the processor architectures.

The successful benchmarks and correct results show that the compiler fulfills the original requirements. However, the compiler leaves much room for further improvements and optimizations.

BIBLIOGRAPHY

[1] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools.* Morgan Kaufmann, 2005.

[2] H. Corporaal, *Microprocessor Architectures: from VLIW to TTA.* John Wiley & Sons, 1997.

[3] J. Hoogerbrugge, "Code generation for transport triggered architectures," Ph.D. dissertation, Delft University of Technology, The Netherlands, 1996.

[4] Department of Computer Systems, Tampere Univ. Tech., "TCE project at TUT," http://tce.cs.tut.fi. [Online]. Available: http://tce.cs.tut.fi

[5] T. U. T. Department of Computer Systems, "TTA codesign environment v1.0 user manual," Project Document, Tampere University of Technology, Finland, 2008.

[6] P. Jääskeläinen, "Instruction Set Simulator for Transport Triggered Architectures," Master's thesis, Department of Information Technology, Tampere University of Technology, Finland, Sep 2005, http://tce.cs.tut.fi/.

[7] V. Korhonen, "Tools for Fast Design of Application-specific Processors," Master's thesis, Department of Information Technology, Tampere University of Technology, Finland, Jan 2009, *See* `http://tce.cs.tut.fi/`.

[8] L. Laasonen, "Program Image and Processor Generator for Transport Triggered Architectures," Master's thesis, Department of Information Technology, Tampere University of Technology, Finland, Apr 2007, *See* `http://tce.cs.tut.fi/`.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

[10] S. S. Muchnick, *Advanced Compiler Design and Implementation.* Morgan Kaufmann, August 1997.

[11] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., 1986.

[12] J. Janssen, "Compiler strategies for transport triggered architectures," Ph.D. dissertation, Delft University of Technology, The Netherlands, 2001.

[13] P. Ienne and R. Leupers, *Customizable Embedded Processors–Design Technologies and Applications*, ser. Systems on Silicon Series. San Mateo, CA: Morgan Kaufmann, 2006.

[14] The LLVM Team, "The LLVM Compiler Infrastructure Project," http://llvm.org. [Online]. Available: http://llvm.org

[15] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation and Optimization*, Palo Alto, CA, March 20–24 2004, p. 75.

[16] Free Software Foundation, "GCC, The GNU Compiler Collection," http://gcc.gnu.org. [Online]. Available: http://gcc.gnu.org

[17] Chris Lattner and Vikram Adve, "LLVM Language Reference Manual," http://llvm.org/docs/LangRef.html. [Online]. Available: http://llvm.org/docs/LangRef.html

[18] C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," in *LCPC'04 Mini Workshop on Compiler Research Infrastructures*, West Lafayette, Indiana, Sep 2004.

[19] Chris Lattner, "TableGen Fundamentals," http://www.llvm.org/docs/TableGenFundamentals.html. [Online]. Available: http://www.llvm.org/docs/TableGenFundamentals.html

[20] Mason Woo and Misha Brukman, "Writing an LLVM Compiler Backend," http://www.llvm.org/docs/WritingAnLLVMBackend.html. [Online]. Available: http://www.llvm.org/docs/WritingAnLLVMBackend.html

[21] A. Oksman, "Machine Object Model," Internal Project Document, Tampere Univ. Tech., Finland, 2004-2005.

[22] A. Cilio and A. Metsähalme, "Program Object Model," Internal Project Document, Tampere Univ. of Tech., Tampere, Finland, 2004-2006.

[23] A. Metsähalme, "Instruction Scheduler Framework for Transport Triggered Architectures," Master's thesis, Department of Information Technology, Tampere University of Technology, Tampere, Finland, Apr 2008, *See* `http://tce.cs.tut.fi/`.

[24] EEMBC, "Denbench 1.0 software benchmark databook," PDF, http://www.eembc.org/TechLit/Datasheets/denbench_db.pdf.