Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2014

MSc THESIS

LLVM-based ρ-VEX compiler

Maurice Daverveldt

Abstract

This thesis describes the development of a LLVM-based compiler for the ρ-VEX processor. The ρ-VEX processor is a runtime reconfigurable VLIW processor. Currently, two compilers exist that target the ρ-VEX processor: a HP-VEX compiler and a GCC-based compiler.
We show that both compilers have disadvantages that are very difficult to fix. Therefore we have built a LLVM-based compiler that targets the ρ-VEX processor. The LLVM-based compiler can be parameterized in a way similar to the HP-VEX compiler. Furthermore, we will present certain optimizations that are new for LLVM-based compilers. These optimizations include a custom machine scheduler that avoids structural and data hazards in the generated binaries.
Finally, we demonstrate the operations of the LLVM-based compiler and compare the performance of generated binaries with the existing compilers. We will show that the LLVM-based compiler exceeds the performance and code quality of the GCC-based compiler. Binaries generated with the HP-VEX compiler outperform those of the LLVM-based compiler.

CE-MS-2014

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

LLVM-based ρ-VEX compiler
asd

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Maurice Daverveldt
born in Leiderdorp, Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

LLVM-based ρ-VEX compiler

by Maurice Daverveldt

Abstract

This thesis describes the development of a LLVM-based compiler for the ρ-VEX processor. The ρ-VEX processor is a runtime reconfigurable VLIW processor. Currently, two compilers exist that target the ρ-VEX processor: a HP-VEX compiler and a GCC-based compiler.

We show that both compilers have disadvantages that are very difficult to fix. Therefore we have built a LLVM-based compiler that targets the ρ-VEX processor. The LLVM-based compiler can be parameterized in a way similar to the HP-VEX compiler. Furthermore, we will present certain optimizations that are new for LLVM-based compilers. These optimizations include a custom machine scheduler that avoids structural and data hazards in the generated binaries.

Finally, we demonstrate the operations of the LLVM-based compiler and compare the performance of generated binaries with the existing compilers. We will show that the LLVM-based compiler exceeds the performance and code quality of the GCC-based compiler. Binaries generated with the HP-VEX compiler outperform those of the LLVM-based compiler.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2014

Committee Members :

Advisor:	dr.ir. Stephan Wong, CE, TU Delft
Chairperson:	dr.ir. K.L.M. Bertels, CE, TU Delft
Member:	dr.ir. A. van Genderen, CE, TU Delft
Member:	dr.ir. Guido Wachsmuth, CE, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

ALU	Arithmetic Logic Unit
AST	Abstract Syntax Tree
CPI	Clocks Per Instruction
DAG	Directed Acyclic Graph
DFA	Deterministic Finite Automaton
DSL	Domain-specific Language
FP	Floating-point
FU	Functional Unit
GCC	GNU Compiler Collection
IDE	Integrated Development Environment
ILP	Instruction Level Parallelism
IPC	Instructions Per Clock
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISD	Instruction SelectionDAG
JIT	Just-in-time compilation
LLVM	Low Level Virtual Machine
MBB	Machine Basic Block
MI	Machine Instruction
OoO	Out-of-Order Execution
PC	Program Counter
RAW	Read After Write
RISC	Reduced Instruction Set Computer
SSA	Single Static Assignment
VEX	VLIW Example
VLIW	Very Long Instruction Word

Acknowledgements

In 2007 I started studying Electronic Engineering and Design at the University of applied sciences Utrecht. After graduating in 2011 I decided to pursue a degree in Computer Engineering at the Technical University Delft. This thesis reflects the new things I have learned since beginning my study at the Electrical Engineering Department.

Firstly I would like to thank my advisor Stephan Wong for giving me the opportunity to work on this subject. Secondly, I would like to thank Rol Seedorf, Anthony Brandon and Joost Hoozemans for their discussions and help during the realization of this project. Their advice has proven to be priceless.

And of course I would like to thank my friends and family for all the support they have given me over the years.

Maurice Daverveldt
Delft, The Netherlands
March 19, 2014

Introduction 1

This thesis will describe the build of a LLVM-based compiler targeting the ρ-VEX processor. In this chapter we describe the motivation for building a new compiler for the ρ-VEX processor. We discuss the history of the ρ-VEX processor and of VLIW processors in general. Furthermore, we are going to see how a LLVM-based compiler can be an improvement over the current solutions that exist.

1.1 Motivation

In 2008, Thijs van As designed the first version of the ρ-VEX processor [4]. This processor uses a VLIW design and is based on the VEX ISA. The VEX ISA is a derivative of the Lx family of embedded VLIW processors [5] from HP/STMicroelectronics. In 2011 the design of the ρ-VEX processor was updated by Roël Seedorf [2]. This new version of the ρ-VEX processor introduced pipelining and forwarding logic. This processor could be parameterized in issue-width, type and number of functional units, number of registers and the presence of forwarding logic.

Around this processor a set of tools has been developed in collaboration with the TU Delft, IBM, STMicroelectronics and other universities. Currently, the ρ-VEX 2.0 tool suite include a synthesizable core, a compiler system, and a processor simulator. IBM has developed a GCC-based VLIW compiler backend.

Very Long Instruction Word (VLIW) [6] processors can execute multiple operations during a single clock cycle. A compiler is required to find parallelism between instructions and to provide scheduling that enables the VLIW processor to execute multiple operations during a single cycle.

Regular RISC type processor, such as the MIPS and ARM processor, contains a single instruction pipeline that executes instructions. Figure 1.1 shows a basic MIPS integer pipeline. By introducing pipelining registers the clock frequency of a processor can be increased because execution of an instruction is broken up into smaller and simpler parts. The RISC pipeline can contain multiple instructions that are in different stages of execution.

Generally speaking, pipelining will decrease the Instructions Per Clock (IPC) rate of a processor because it is very difficult to use all the stages of the pipeline all the time. Pipelining introduced hazards where situations will occur that force the pipeline to wait until a certain execution has finished executing. This wait cycle decreases the IPC of the processor and in turn decreases the performance. Special hardware has been developed, such as forwarding units, branch predictors, and speculative execution that will try to increase the CPI to a value that approaches 1.0 [1].

If a higher than 1.0 IPC is desired multiple instructions need to be executed during a single clock cycle. Machines that can execute multiple instructions are called multi-issue

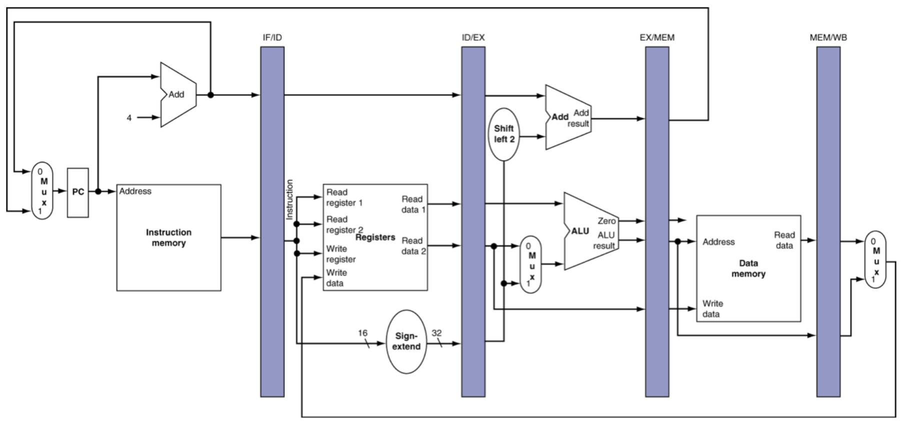

Figure 1.1: MIPS pipeline [1]

machines. These types of processors use special hardware to find dependencies between instructions and to determine which instructions can be executed in parallel. These techniques include Tomasulo's algorithm [7] for *Out of Order* execution and register renaming. Most modern processors use these techniques to increase performance.

Finding dependencies between instructions becomes increasingly complex when the issue-width of machines is increased. The Pentium 4 processor [8]demonstrated the limitations of further ILP extraction in a spectacular way. It used a 20-stage pipeline [9] with seven functional units. It operated on RISC-like micro-ops instead of x86 instructions and could handle 50 in-flight instructions at a time. The amount of silicon and energy that was dedicated to finding and executing ILP made the Pentium 4 processor very inefficient. The clock frequency increase that Intel expected the hyper pipelined processor (6-7 GHz) to deliver never materialized and the Pentium 4 Netburst architecture was dropped for a much simpler architecture [10].

In [11] the actual limitations of ILP extraction in hardware has been demonstrated and it has been shown that other techniques need to be used to find and execute more ILP.

VLIW processors differ from multi-issue machines in that parallelism is found during compile-time instead of during run-time [12] [13]. This results in a processor that can be made significantly simpler because the ILP extraction algorithms do not need to be implemented and because dependency checking is not required during run-time. Additional ILP can also be found with the compiler because the compiler has got a higher-level view of the code that is to be executed. Optimizations such as swing modulo scheduling and loop vectorization are nearly impossible to achieve in hardware because the higher-level structure of a program is no longer available. A compiler can interpret the higher-level structure of a program and optimize the output for better scheduling.

The origins of the VEX ISA can be traced to the company Multiflow and John Fisher, one of the inventors of VLIW processors at Yale University [6]. Multiflow designed a computer that used VLIW processors to execute instructions up to 1024-bits in size. Along with these computers Multiflow also designed a compiler system that used trace based scheduling to extract ILP from programs [14]. Reportedly the code base for the

Multiflow compiler has been used in modern compiler such as Intel C Compiler (ICC) and HP-VEX compiler because of the robustness and the amount of ILP that could be exposed by the compiler [15].

John Fisher has designed the VEX ISA as an example of VLIW type processors [16]. His work includes the design of an ISA, processor design, and a compiler system that generates code for this processor. Thijs van As developed a processor called ρ-VEX that is binary compatible with the VEX processor [4].

Currently two different compilers exist that target the ρ-VEX processor: the HP-VEX compiler [17] and a GCC port developed by IBM. We will show that both existing compilers are not optimal and that a new compiler is required for the ρ-VEX project. Further we will present a LLVM based compiler that targets the ρ-VEX processor with performance and features similar to the HP-VEX compiler.

1.2 Problem statement

Currently, both the HP-VEX and GCC compilers can be used to generate code for the ρ-VEX processor. Both compilers have got a number of advantages and disadvantages that will be explored. The compilers will be judged on the following properties: Code quality, support, languages support, backend supported and customization possibilities.

- **Code quality:** The code a compiler produces must be correct and fast. Compilation for VLIW type processors is complex and a compiler is required that is able to produce code of high quality that uses the features offered by the ρ-VEX processor in the best possible way.
- **Support:** Compiler development becomes easier if an active community is available that can help with support. Furthermore, an active community also ensures the compiler can gain new features.
- **Front-end languages:** The compiler is more flexible to the end-user when a large number of front-end languages are supported. This allows the end-user to choose the language that he or she is most familiar with for development of applications.
- **Customization:** The ρ-VEX processor can be reconfigured using parameters. The code that is generated by the compiler needs to reflect the hardware configuration of the ρ-VEX processor.

Based on these properties we have analyzed the HP-VEX and GCC-based compiler. HP-VEX:

- **Code quality:** Excellent code quality and ILP extraction.
- **Support:** Bad, no active community.
- **Front-end:** Bad, only support for C.
- **Back-end:** Not applicable since compiler is specifically targeted to one architecture.

- **Customization:** Customization possible through machine description. Further research on optimization strategies are not possible because compiler is proprietary and closed source. Because of this expanding the functionality of the compiler is impossible.

GCC:

- **Code quality:** The ρ-VEX backend for GCC has not been optimized and the quality of the code is quite low. Performance of GCC executables is lower then code compiled by the HP-VEX compiler. Some programs do not function correctly when compiled by GCC.
- **Support:** There is a very active development community around GCC itself [18]. Unfortunately, the support for the ρ-VEX is not good because the ρ-VEX branch is maintained privately.
- **Front-end:** GCC supports a large number of programming languages including C, C++, Fortran and Java
- **Customization:** Because GCC is open source the compiler can be customized to support new passes, optimizations and instructions. The ρ-VEX backend currently only supports a 4-issue ρ-VEX processor. Furthermore, the internal workings of GCC are complex and poorly documented. Different parts of the compiler are linked in a complex way and it is very difficult to obtain a general overview on how the compiler operates. Because of the complexity it is difficult to add new functionality to the GCC compiler.

The comparison shows that both the HP-VEX and GCC compilers have serious disadvantages. The fact that HP-VEX cannot be customized excludes it from further development for the ρ-VEX project. Bringing the GCC compiler performance and features up to the same level as HP-VEX will be very difficult because of the complexity involved with GCC development.

In 2000, the LLVM project [19] has been started with the goal of replacing the code generator in the GCC compiler. LLVM provides a modern, modular design and is written in C++. Originally, the GCC front-end was used to translate programs into LLVM compatible intermediate representation. Around 2005, the Clang project was started which aimed to replace the GCC front-end with an independent front-end that supports C, C++ and ObjC. Currently the LLVM-based compiler offers performance that approaches GCC but offering a significant improvement in terms of modularity, ease of development and "hackability". In addition, the LLVM compiler can also be used to target different architectures such as GPU's and VLIW based processors.

1.2.1 Previous works

Currently the LLVM compiler has support for the Qualcom Hexagon VLIW processor [20]. In [21] it is mentioned that it took two engineers 23 days to get a LLVM-based back-end working. Within 107 Calendar days they were able to achieve 87% performance of GCC.

[22] documents the implementation of a LLVM backend for the TriCore processor. They conclude that their TriCore backend could emerge as a serious alternative to the current compilers that target the TriCore processor.

A similar project that aims to create a TriCore backend for GCC is presented in [23]. While they were able to complete the backend they concluded that porting GCC is a hard task and that it requires intricate knowledge of the GCC internals.

1.3 Goals

The main goal of this thesis is to develop a new compiler for the ρ-VEX system. The compiler will be based on the LLVM compiler. The new compiler should have the following characteristics:

- **Open source:** The compiler should be open source so the compiler can be customized and used for future research.
- **Code quality:** A new compiler should provide a significant improvement in terms of performance, code size and resource utilization.
- **Reconfigurability:** Characteristics of the ρ-VEX processor should be reconfigurable during run-time.

1.4 Methodology

The following steps need to be completed for successful implementation of a ρ-VEX LLVM compiler.

- Research ρ-VEX and VEX platform
- Research LLVM compiler framework
- Build LLVM-based VEX compiler with following features:
  - 4-issue ρ-VEX processor
  - Code generation
  - Assembly emitter
- Add support for reconfigurability:
  - VEX machine description
  - Reconfigure LLVM during runtime
- Optimize performance:
  - Instruction selection
  - Hazard recognizer
  - Register allocator

1.5 Thesis overview

The thesis is organized as follows. In Chapter 2 we will discuss the architecture of the ρ-VEX processor and the workings of the LLVM compiler suite. This chapter will demonstrate the supported instructions, run-time architecture, and show the general architecture of the ρ-VEX processor. The chapter will also show how the LLVM compiler operates and what steps are involved during compilation.

In Chapter 3 will discuss how the ρ-VEX compiler was implemented. We will show how code is transformed from the LLVM Intermediate Representation (IR) into a ρ-VEX specific assembly language. In addition, we will also discuss new functionality that has been added to the LLVM compiler.

Chapter 4 will discuss how the performance of the LLVM compiler has been optimized. Compilation problems that have been found are demonstrated and we will show how these problems have been resolved to increase performance of the binaries.

Chapter 5 will explore the performance of the new compiler. Performance will be compared to existing compilers in terms of issue-width.

A conclusion and recommendations for future research is presented in Chapter 6.

Background 2

In this chapter we will explore the background of the VEX system and the LLVM compiler. This chapter will show the basic design of the ρ-VEX processor and how the ρ-VEX processor operates. Further, we will also demonstrate the design of the LLVM compiler framework and how code is transformed from a high-level language such as C/C++ to a target specific assembly language.

2.1 VEX System

The ρ-VEX processor is based on the VEX ISA [4]. The processor uses a VLIW architecture and is designed to serve as both a application-specific processor and a general-purpose processor. During synthesis the core can be reconfigured to alter the issue-width, the amount of physical registers, the type of functional units, and other parameters.

The VEX ISA defines hardware operations as syllables. An instruction is defined as a set of multiple syllables. The VEX operations are similar to 32-bit RISC operations.

2.1.1 Architecture

The architecture of the ρ-VEX core is depicted in Figure 2.1 [24]. The core uses a five-stage pipelined design. There are multiple functional units with different functionalities, such as ALUs, multipliers, load / store units, and branch units.

The register file consists of 64 32-bit general-purpose registers. These registers can generally be targeted by any instruction. In addition to the general-purpose registers there also exists 8 1-bit branch registers. These registers are used by the branch operations to determine if a branch should occur.

The pipeline uses five stages to execute an instruction. For example, consider a ρ-VEX processor with a 4-issue organization:

- **Fetch stage:** An instruction is selected from the instruction memory using the Program Counter (PC) or the Branch Target address. Note that 1 instruction contains four different operations.

- **Decode stage:** Each operation is decoded in parallel and the needed registers are read from the register file.

- **Execute 1 stage:** In this stage, the functional units produce results for ALU operations. In addition this stage is also used to write to the data memory when Store operations are executed. The ρ-VEX processor uses $16*32$-bit multipliers that require two stages to produce a result. In this stage the multiplication operations are started.

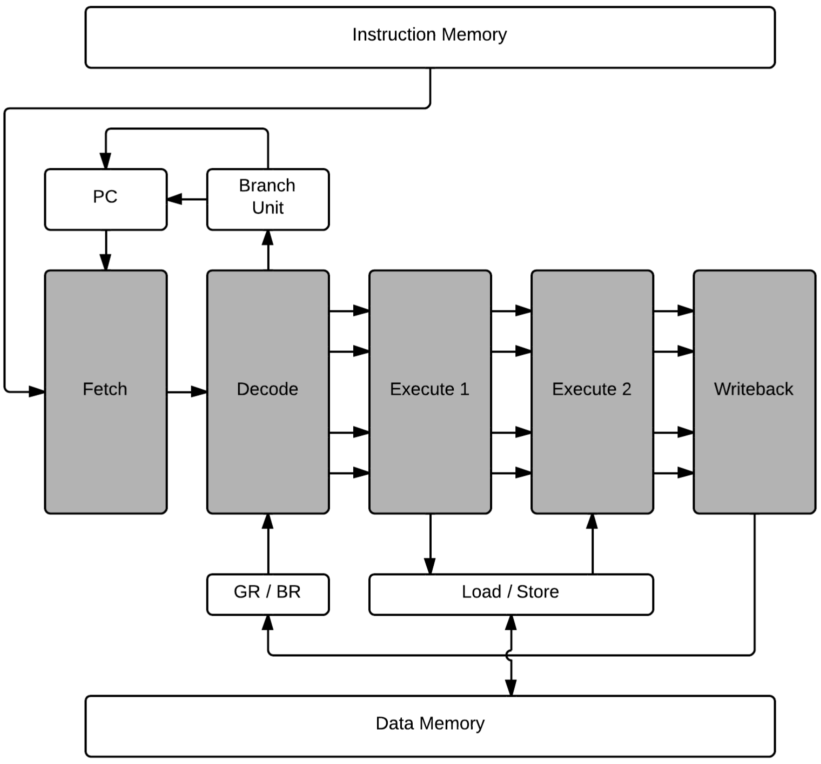

Figure 2.1: ρ-VEX architecture [2]

- **Execute 2 stage:** This stage produces results for multiply operations. In addition this stage reads data from the data memory when load operations are executed.
- **Writeback stage:** In this stage results that have been generated in the previous stages are committed to the register file.

The ρ-VEX processor uses a bypass network to forward operations to other pipeline stages when needed [2].

2.1.2 ISA

The assembly format for VEX instructions is shown in Figure 2.2 [3]. The destination of an operation is to the left of the "=" sign, while the source operands are listed on the right-hand side. On the right-hand side both registers and immediate values can be used as source operands. The ρ-VEX processor does not support multiple clusters and each instruction is executed on cluster 0.

The ρ-VEX processor supports 32-bit immediate values through an operations borrowing scheme. Each instruction can supports 8-bit immediate values but if larger values are required the adjacent operation is used to store the upper 24-bits of the immediate

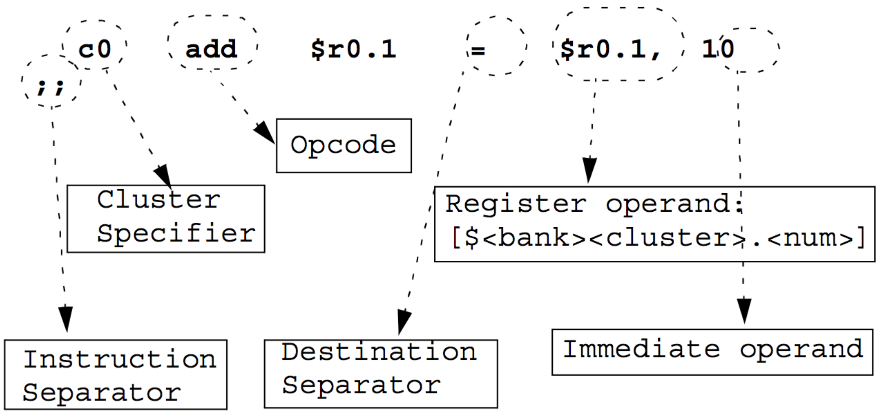

Figure 2.2: ρ-VEX instruction format

value. This means that when using large immediate value the amount of operations that can be executed decreases.

Multiple classes of ρ-VEX instructions exists with the following properties:

- **Integer arithmetic operations:** These operations include the traditional RISC-style instructions such as `ADD`, `SUB`, `AND`, and `OR`.
- **Multiplication operations:** The VEX ISA defines multiple multiplication operations that use the built-in $16 * 32$-bit multiplier. Operations include for example: Multiply Low 16 * Low 16, etc.
- **Logical and Select operations:** These operations are used to compare two registers to each other or to select between two values based on the result of a branch register. Operations include: `CMPEQ`, `CMPNEQ`, etc.
- **Memory operations:** Operations that load and store data from the data memory. Operations exist to store and load operands of different sizes such as `LDW`, `LDH` and `LDB`.
- **Control operations:** These operations are used to control the Program Counter of the ρ-VEX processor. Operations include: `GOTO`, `CALL`, `BR`, and `RETURN`.

2.1.3 Run-time architecture

The ρ-VEX Run-Time architecture defines the software conventions that are used during compilation, linking and execution of ρ-VEX executables. ρ-VEX programs are executed in a 32-bit environment where integers, longs, and pointers are 32-bit values.

The following ρ-VEX register classes are used:

- **Scratch registers:** Caller-saved registers that are destroyed during function calls.
- **Preserved registers:** Callee-saved registers that must not be destroyed during procedure calls.

Register	**Class**	**Description**
`$r0.0`	Constant	Constant register 0
`$r0.1`	Special	**Stack-pointer:** Holds the limit of the current stackframe. The SP is preserved across function calls.
`$r0.2`	Scratch	**Struct return pointer:** If a function returns a struct or union the register contains the memory adres of the value being returned.
`$r0.3-$r0.10`	Scratch	**Arguments and return values:** Arguments that do not fit in the registers are passed using the main memory.
`$r0.11-$r0.56`	Scratch	Caller-saved scratch registers.
`$r0.57-$r0.63`	Preserved	Callee-saved registers that need to be preserved across function calls.
`$l0.0`	Special	**Link register:** Used to store the return adres when a function call is performed.
`$pc0.0`	Special	**Program Counter**
`$b0.0-$b0.7`	Scratch	**Branch registers:** Caller-saved registers.

Table 2.1: ρ-VEX Register usage [3]

- **Constant registers:** Contains a value that cannot be changed.
- **Special registers:** Used during call / return operations.

The 2.1 described the properties of all the available ρ-VEX registers.

2.2 LLVM Compiler infrastructure

LLVM is based on the classic three-stage compiler architecture depicted in figure 2.3. The compiler uses a number language-specific frontends, an optimizer and target-specific backends. Each module consists of a number of generic passes that are used to transform the code. This modular design enables compiler designers to introduce new passes and parts of the compiler without having to change the existing framework. Support for a new processor can be added by building a new back-end. The existing frontend and optimizer can be reused for the new compiler.

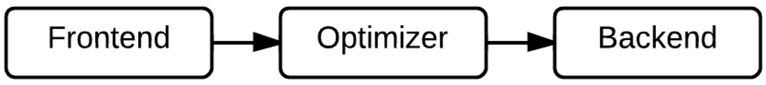

Figure 2.3: Basic compiler structure

The frontend is used to transform the plain text source code of a program into an intermediate representation that will be used during compilation process. This transformation is achieved by performing the following steps:

1. **Lexical analysis:** Break input into individual tokens.
2. **Syntax analysis:** Using a grammar, the sequence of tokens is transformed into a parse tree which represents the structure of the program. Clang uses a handwritten recursive descent parser for this transformation
3. **Semantic analysis:** Semantic information is added to the parse tree, type checking is performed, and a symbol table is built.

The resulting abstract syntax tree (AST) is transformed into LLVM IR and passed to the optimizer and backend of the compiler. These parts of the compilation process are completely language agnostic and do not require any other information from the backend.

The optimizer is used to analyze and optimize the program. Optimization such as dead code elimination and copy propagation are performed during this phase but also more advanced operations that extract ILP, such as loop vectorization [25], can be enabled.

The back-end optimizes and generates code for a specific architecture. The LLVM IR is transformed into processor specific assembly instructions. In addition the backend also allocates registers, and schedules the instructions.

2.2.1 Current frontends

The modular design of LLVM enables the compiler to be used as a part of the existing GCC compiler. For example, the dragonegg GCC plugin [26] is designed to replace the GCC code generator and optimizer with the LLVM backend. This would enable LLVM to be able to use the existing GCC based frontends and supported languages.

Clang has been developed to allow LLVM to operate independently of GCC. Clang is a frontend supporting C, C++, and ObjC. The frontend is designed to be closely integrated with the Integrated Development Environment (IDE) allowing more expressive diagnostic messages. In addition, Clang also aims to provide faster compilation and lower memory usage [27].

2.2.2 LLVM IR

The frontend transforms a source code into the LLVM Intermediate representation (LLVM IR). The LLVM IR is used to represent a high level language cleanly in a target independent way and is used during all phases of compilation. Instructions are similar to RISC instructions and can use three operands. Control flow instructions and type specific load/store instructions are used and an infinite amount of registers are available in Single Static Assignment (SSA) form. The LLVM IR is available in three different forms: human readable text, binary form, and an in-memory form [28].

The LLVM IR is designed to expose high-level information for further optimization. Examples of high-level information include dataflow analysis using the SSA form, control-flow graphs, language independent type information and explicit use of pointer arithmetic.

Primitives such as voids, floats and integers are natively supported in the LLVM IR. The bit width of the integers can be defined manually. Pointers, arrays, structures and functions are derived from these basic types. The operations that are supported in LLVM IR are contained in the Instruction Selection DAG (ISD) namespace.

Object-oriented constructs such as classes and virtual methods are not natively supported but can be built using the existing type system. For example, a C++ class can be represented by a struct and a list of methods.

The SSA-based dataflow form allows the compiler to efficiently perform code optimizations such as dead code elimination and constant propagation.

Figure 2.1 depicts an example program in C. The equivalent LLVM IR representation is depicted in figure 2.2.

```c
int main() {
  int sum = 1;

  while(sum < 10)
  {
    sum = sum + 1;
  }
  return sum;
}
```

Listing 2.1: C example program

```llvm
define i32 @main() nounwind ssp uwtable {
  %1 = alloca i32, align 4
  %sum = alloca i32, align 4
  store i32 0, i32* %1
  store i32 1, i32* %sum, align 4
  br label %2

; <label>:2                                       ; preds = %5, %0
  %3 = load i32* %sum, align 4
  %4 = icmp slt i32 %3, 10
  br i1 %4, label %5, label %8

; <label>:5                                       ; preds = %2
  %6 = load i32* %sum, align 4
  %7 = add nsw i32 %6, 1
  store i32 %7, i32* %sum, align 4
  br label %2

; <label>:8                                       ; preds = %2
  %9 = load i32* %sum, align 4
  ret i32 %9
}
```

Listing 2.2: LLVM Intermediate representation

2.2.3 Code generation

During code generation the optimized LLVM IR is translated into machine-specific assembly instructions. The modular design of LLVM enables generic algorithms to be used for this process.

A backend is described in a domain-specific language (DSL) called `tablegen`. The `tablegen` files describe properties of a backend such as available instructions, registers, calling convention and pipeline structure. During compilation of LLVM the `tablegen` files are converted into a C++ description of the backend. `tablegen` has been specifically designed to describe the backend structure in a flexible and generic way. Common features can be more easily described using `tablegen`. For example the `add` and `sub` instruction are almost identical and using `tablegen` can be described in a more generic way. This results in less repetition and reduces the chance of error in the backend description.

Because of the generic description of the backend large amount of code can be reused by each backend. Algorithms such as register allocation and instruction selection operate on the generic `tablegen` descriptions and do not require target specific hooks to operate correctly. An additional advantage of this approach is that multiple algorithms are available to achieve certain functionality. For example, LLVM offers the developer a choice between four different register allocation algorithms. Each algorithm has a number of advantages and disadvantages and the developer can choose between an algorithm which matches the target processor best.

At the moment not all parts of the backend can be described in `tablegen` and hand written C++ code is still needed. As LLVM matures more parts of the backend description should be integrated into the backend.

Figure 2.4 shows the basic code generation process. Each block can consist of multiple LLVM passes. For example the instruction selection phase consists of multiple passes that transform the input LLVM IR into a DAG that only contains instructions and types that are supported by the target processor.

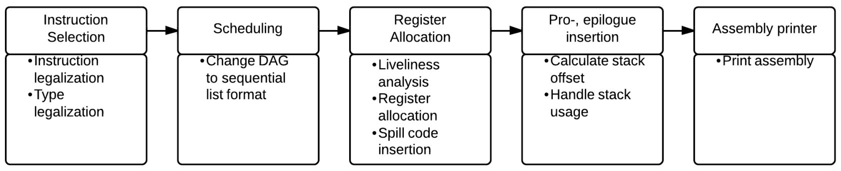

Figure 2.4: Basic codegeneration process

2.2.4 Scheduling

The LLVM compiler uses basic blocks to schedule instructions. A basic block is a block of code that has exactly one entry point and one exit point. This means that no jump instruction exists with a destination in the block.

LLVM uses the `MachineBasicBlock` (MBB) class to represent a Basic Block. A MBB contains a list of `MachineInstr` instances. The `MachineInstr` class is an abstract way to represent instructions for the target processor.

Multiple MBB are used to create a `MachineFunction` instance. The `MachineFunction` class is used to represent a LLVM IR function. In addition to a list of MBB the MachineFunction also contains references to the `MachineConstantPool`, `MachineFrameInfo`, `MachineFunctionInfo`, and `MachineRegisterInfo`. These classes keep track of target specific function information such as which constants are spilled to memory, the objects that are allocated on the stack, target specific function information, and which registers are used.

2.2.5 Current backends

The current version of the LLVM compiler supports the following processor architectures:

- **ARM:** RISC type processor designed for use in embedded systems.
- **AArch64:** Targets ARMv8 ISA that has support for 64-bit architectures.
- **Hexagon:** 32-bit DSP processor developed by Qualcomm. Targets a VLIW type processor
- **MIPS:** One of the original RISC type processors [29].
- **MBlaze:** Derivative of the MIPS processor developed by Xilinx for use in FPGAs.
- **MSP430:** 16-bit processor developed by Texas Instruments.
- **NVPTX:** CUDA backend targeting NVIDIA GPUs.
- **PowerPC:** RISC type processor developed by IBM.
- **R600:** Backend that targets R600 AMD GPUs.
- **Sparc:** One of the original RISC type processors [30].
- **SystemZ:** Processor used in IBM Mainframes.
- **X86:** General purpose processor.
- **XCore:** 32-bit RISC type processor.

Backends that target other architectures also exist but these are not included in the distribution of the LLVM compiler and are not actively maintained by the LLVM community. The Hexagon, NVPTX, and R600 backends are of special interest because these backends target VLIW processors or massive parallel systems such as GPUs. The Hexagon processor demonstrates that it is possible to build a LLVM backend that targets a general purpose VLIW type processor.

2.3 Verification

Verification of the LLVM compiler is extremely important. Performance of the generated binaries is irrelevant if only half of the binaries produce a correct result. The LLVM compiler will be verified by simulating the generated binaries.

Two simulators are available: XSTsim and Modelsim. XSTsim is an ISA simulator that can simulate a 4-issue ρ-VEX processor. Output of the simulator is customizable and the simulator is fast enough to simulate large executables. Modelsim is used to perform a complete functional simulation of the ρ-VEX processor. The Modelsim simulation provides the highest accuracy because the hardware description files of the ρ-VEX processor are used during simulation. The disadvantage of using Modelsim is the performance. Simulation of an executable will take a long time because the complete processor is simulated.

Writing test programs that generate certain instruction sequences will be used to verify the compiler. The output of these test programs will be compared to the expected result to check for errors in the backend.

Writing test programs that have a high coverage of all the possible output patterns is impossible. To further verify the compiler we will compile benchmark programs that simulate certain. The output of the benchmarking programs will be compared to expected outputs to check for errors in the compiler.

2.4 Conclusion

This chapter presented the basic design of the ρ-VEX processor and of the LLVM compiler framework. The ρ-VEX processor is a VLIW-type processor that uses RISC like instructions to operate. We presented the basic design of the processor, the instructions that are supported, register properties, and the run-time architecture has been discussed. This information will be used during implementation of the LLVM-based compiler.

We also discussed the basic working of the LLVM compiler framework. Building a ρ-VEX backend for the LLVM compiler is feasible because the current version of the LLVM compiler already targets a VLIW processor.

Finally, we discussed how the LLVM compiler will be verified. The verification step is extremely important to determine whether the binaries that are generated work correctly.

Implementation 3

The previous chapter demonstrated the architecture of the ρ-VEX processor and of the LLVM compiler. We have also shown that the LLVM compiler currently has support for a VLIW type processor. In this chapter we will show how the LLVM-based backend for the ρ-VEX processor has been implemented.

3.1 Tablegen

LLVM uses a domain-specific language (DSL) called `tablegen` to describe features of the backend such as instructions, registers, and pipeline information.

`tablegen` uses a object-oriented approach to describe functionality. Information is described in classes and definitions that are called *records*. Inheritance is supported so classes can derive information from superclasses. In addition, multiclasses can be used to instantiate multiple abstract records at once.

The `tablegen` tool aims to provide a flexible way to describe processor features. The processor instructions are be described as follows:

A class is created that represents an abstract instruction. The class will describe information that is of direct importance to code generation such as opcode, register usage and immediate values but also information that is needed during the code generation process such as liveness information, instruction patterns, and scheduling information.

```
class rvexInst<dag outs, dag ins, string asmstr, list<dag> pattern,
               InstrItinClass itin, Format f, CType type>: Instruction
{
}
```

Listing 3.1: Base ρ-VEX instruction

The `rvexInst` class is used for all type of instructions that are supported by the ρ-VEX processor. Multiple instructions with common feature such as `add`, `sub`, and `and` instructions can be described in subclasses that inherit from the `rvexInst` class. For example, the class `ArithLogicR` holds arithmetic instructions that use three register operands. The class describes common features such as the instruction string format and the instruction pattern that are used during instruction selection.

```
class ArithLogicR<string instr_asm, SDNode OpNode,
                  InstrItinClass itin, RegisterClass RC, bit isComm = 0, CType ↩
                      type>:
  rvexInst <(outs RC:$ra), (ins RC:$rb, RC:$rc),
     !strconcat(instr_asm, "\t$ra = $rb, $rc"),
     [(set RC:$ra, (OpNode RC:$rb, RC:$rc))], itin, type>
{
}
```

Listing 3.2: Class representing three-operand instructions

The following code shows how to define instructions that inherit from the `ArithLogicR` class. The instruction is defined as using the `ArithLogicR` class with certain parameters that match the instruction properties. These properties include instruction string, LLVM IR opcode, and other information that is needed during compilation.

```
def ADD          : ArithLogicR<"add ", add, IIAlu, CPURegs, 1, TypeIIAlu>;
```

Listing 3.3: `add` instruction

Figure 3.1 displays an example of how individual instructions inherit from the high-level classes. The final class is a `rvexInstr` that contains a description of all the available ρ-VEX instructions.

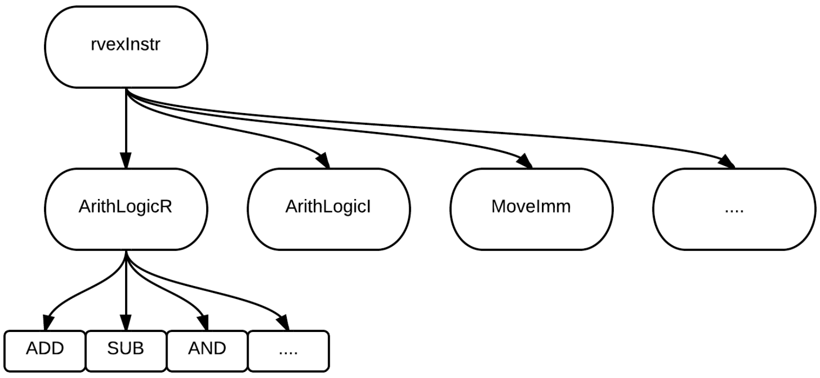

Figure 3.1: `tablegen` instructions

`tablegen` provides for a very flexible way to describe backend functionality. The existing LLVM backends use `tablegen` in a variety of ways which best match the target processor.

The *tblgen* tool is used to transform the `tablegen` input files into C++. The resulting C++ files contain enums, structs and arrays that describe the properties. The instruction selection part is transformed into imperative code that is used by the backend for pattern matching.

3.1.1 Register definition

LLVM uses a predefined class `register` to handle register classes. All ρ-VEX registers are derived from this empty class. The `rvexReg` class is used to define all type of ρ-VEX registers.

```
class rvexReg<string n> : Register<n> {
  field bits<7> Num;
  let Namespace = "rvex";
}
```

Listing 3.4: Register class definition

The `rvexReg` class is used to define the general purpose registers and the branch registers.

```
class rvexGPRReg<bits<7> num, string n> : rvexReg<n> {
  let Num = num;
}

class rvexBRReg<bits<7> num, string n> : rvexReg<n> {
  let Num = num;
}
```

Listing 3.5: ρ-VEX register types

Each physical register is defined as an instance of one of these classes. For example, `r0.5` is defined as follows. The register is associated with a register number, a register string, and a dwarf register number that is used for debugging.

```
def R5 : rvexGPRReg< 5, "r0.5">, DwarfRegNum<[5]>;
```

Listing 3.6: ρ-VEX register

The physical registers are divided in two register classes for the general purpose registers and for the branch registers. The register classes also define what type of value can be stored in the physical register.

```
def CPURegs : RegisterClass<"rvex", [i32], 32,
  (add
    (sequence "R%u", 0, 63),
    LR, PC
  )>;

def BRRegs   : RegisterClass<"rvex", [i32], 32,
  (add
    (sequence "B%u", 0, 7)
  )>;
```

Listing 3.7: ρ-VEX register classes

The branch registers have been defined to also use 32-bit values even though in reality the branch register is only 1-bit wide. This has been done because LLVM had trouble identifying the correct instruction patterns for compare instructions. The ρ-VEX compare instructions can produce results in both the `CPURegs` and the `BRegs` as illustrated in the following example:

```
<1 bit>BRRegs = Operation, <32 bit>CPURegs, <32 bit>CPURegs
<32 bit>CPURegs = Operation, <32 bit>CPURegs, <32 bit>CPURegs
```

The LLVM compiler is unaware that when the compare instruction is used to define a 32-bit result only the lowest bit will be set. The compiler tried to resolve this issue by inserting `truncate` and `zero_extend` instructions even though this is not required. This has been solved by implementing the `BRRegs` as 32-bit wide so LLVM will not insert truncate and extend instructions when operating on these type of instructions.

3.1.2 Pipeline definition

`tablegen` can be used to describe the architecture of the processor in a generic way. LLVM will schedule an instruction to a processor functional unit during the scheduling pass. The following code describes the available functional units for a 4-issue a ρ-VEX processor.

```
def P0 : FuncUnit;
def P1 : FuncUnit;
def P2 : FuncUnit;
def P3 : FuncUnit;
```

Each instruction is associated with an instruction itinerary. An instruction itinerary is used to group scheduling properties of instructions together. The ρ-VEX processor uses the following instruction itineraries.

```
def IIAlu               : InstrItinClass;
def IILoadStore         : InstrItinClass;
def IIBranch            : InstrItinClass;
def IIMul               : InstrItinClass;
```

The functional units and instruction itineraries are used to describe the properties of the ρ-VEX pipeline. The scheduling properties are derived from a description of an instruction stage with certain properties and the associated instruction itinerary. These properties include the *cycle count*, that describes the length of the instruction stage, and the functional units that can execute the instruction. The following itinerary describes a ρ-VEX pipeline with four functional units. Each functional unit is able to execute every instruction except for load/store instructions. Only `P0` is able to execute load/store instructions. These instructions take two cycles to complete.

```
def rvexGenericItineraries : ProcessorItineraries<[P0, P1, P2, P3], [], [
  InstrItinData<IIAlu              , [InstrStage<1,  [P0, P1, P2, P3]>]>,
  InstrItinData<IILoadStore        , [InstrStage<2,  [P0]>]>,
  InstrItinData<IIBranch           , [InstrStage<1,  [P0, P1, P2, P3]>]>,
  InstrItinData<IIIMul             , [InstrStage<1,  [P0, P1, P2, P3]>]>,
]>;
```

The machine model class is used to encapsulate the processor itineraries and certain high-level properties such as issue-width and latencies.

```
def rvexModel : SchedMachineModel {
  let IssueWidth = 2;
  let Itineraries = rvexGenericItineraries;
}
```

3.1.3 Other specifications

`tablegen` is also used to describe other properties of the target processor. LLVM has stated as goal to move more parts of the backend description to the `tablegen` format because `tablegen` offers such a flexible implementation. At the moment `tablegen` is also used to implement:

- **Calling convention:** Describes the registers that are used to pass arguments between functions, the return registers, the callee-saved registers, the caller-saved registers and the reserved registers.
- **Subtarget features:** Target-specific features can be defined using the subtarget description.

3.2 Code generation

To understand how the compiler changes code from the LLVM IR representation to VEX assembly instruction it is necessary to understand how the code generation process works. The code generation process is divided into multiple steps, called passes, which are performed in order. Directed acyclic graphs (DAG) are used to represent the LLVM IR during the code generation process.

3.2.1 Instruction transformation

The instruction selection phase is completed in the following steps

- **Build initial DAG:** Transform the LLVM IR into a DAG that contains illegal types and instructions. The initial DAG is a one-to-one representation of the LLVM IR code.

- **Legalize instructions:** Illegal instructions are expanded and replaced with legal instructions that are supported by the target processor.

- **Legalize types:** Transform the types used in the DAG to types that are supported by the target processor

- **Instruction selection:** The legalized DAG still contains only LLVM IR instructions. The DAG is transformed to a DAG containing target-specific processor instructions.

3.2.2 Instruction Lowering

The `rvexISelLowering` class gives a high-level description of the instructions that are supported by the target processor. The class can describe how the compiler should handle each LLVM IR instruction using four parameters: Legal, Expand, Promote or Custom. The default option is Legal, which implies that the LLVM IR instruction is natively supported by the target processor.

3.2.2.1 Expanded instructions

The Expand flag is used to indicate that the compiler should try to expand the instruction into simpler instructions. For example consider the LLVM IR `UMUL_LOHI` instruction. This instruction multiplies two values of type `iN` (for example 32-bit) and returns a result of type `i[2*N]` (64-bit). Through expansion this instruction will be transformed into two multiply instructions that calculate the low part and the high part separately.

```
setOperationAction(ISD::UMUL_LOHI,  MVT::i32, Expand);
```

3.2.2.2 Promote instruction

Some instruction types are not natively supported and the type should be promoted to a larger type that is supported by the target processor. This feature is useful for supporting logical operations on Boolean functions. The following operation transforms an AND instruction that operates on a boolean value to a larger type.

```
setOperationAction(ISD::AND,         MVT::i1, Promote);
```

3.2.2.3 Custom expansion

There are some instructions that cannot be expanded automatically by the compiler. To support these instructions the instruction expansion can be defined manually. For example consider the `MULHS` instruction that multiplies two numbers and returns the high part.

```
setOperationAction(ISD::MULHS,       MVT::i32, Custom);
```

When a `MULHS` instruction is parsed the compiler will execute a function that describes the sequence of operations to lower this instruction. This sequence of instructions is implemented in the `LowerMULHS` function of the `rvexISelLowering` class. The `LowerMULHS` function is used to manually traverse the DAG and insert a sequence of instructions to support the operation.

For each instruction that requires custom lowering a `LowerXX` function has been defined.

3.2.3 Instruction selection

After instruction lowering the DAG contains LLVM IR operations and types that are all supported by the target processor but the DAG still contains only LLVM IR operations and no target-specific operations. The `rvexISelDAGToDag` class is used to match LLVM IR instructions to instructions of the target processor. The bulk of this class is generated automatically from the `tablegen` description but instructions can also be matched manually.

3.2.4 New instructions

The ρ-VEX processor supports some instructions that have no equivalent LLVM ISD operation. These instructions include `divs`, `addcg`, `min`, `max`, and others. Two stages are required to add support for these operations:

1. **Extend ISD namespace:** The new instructions will be added to the ISD namespace. This means that the LLVM IR will be *extended* with the new instructions.

2. **Instruction lowering:** Describe when these instruction should be inserted in the LLVM IR. For example, lowering of the LLVM IR `div` instruction uses a custom lowering function to describe the algorithm that uses the ρ-VEX `divs` instruction.

3. **Instruction matching:** New pattern matching rules need to be defined that map a custom instruction from the extended ISD namespace to the final target-specific ρ-VEX instructions.

For this example we are going to consider the ρ-VEX `divs` instruction.

3.2.4.1 Extend ISD namespace

The ISD namespace can be extended with target-specific operations by defining the instruction type and the instruction name. Because the `divs` instruction uses five operands a custom instruction type will be used. The following code shows the definition for the custom instruction type. This type describes an instruction that produces two results and consumes three operands.

```
def SDT_rvexDivs           : SDTypeProfile<2, 3
                               [SDTCisSameAs<0, 2>,
                               SDTCisSameAs<0, 3>,
                               SDTCisInt<0>, SDTCisVT<0, i32>,
                               SDTCisSameAs<1,4>,
                               SDTCisInt<1>, SDTCisVT<1, i1>]>;
```

The next step involves defining the name of the custom instruction. The instruction is defined as a custom `SDNode` and uses the instruction type that has been defined earlier. This operation expands the LLVM ISD namespace with custom operations that are only available in the ρ-VEX backend.

```
def rvexDivs               : SDNode<"rvexISD::Divs", SDT_rvexAddc>;
```

3.2.4.2 Instruction lowering

The `rvexISelLowering` class is extended with functions that produce the new ISD operation. For this example LLVM IR `div` instruction is custom lowered in the `LowerDIVS` function. In this function an algorithm is implemented that uses the new `divs` SDNode.

After instruction lowering a DAG will have been produced that contains only legal ISD operations and the new ISD operations that have been defined earlier.

3.2.4.3 Instruction selection

The following code describes the instruction class that is used by the `divs` instruction. This class describes the instruction string, register class properties and certain scheduling properties of this instruction. Note that the instruction pattern is empty because the current version of `tablegen` has no support for instructions that produce multiple results. A custom pattern matching function will need to be implemented in the `rvexISelDAGToDAG` class.

```
class ArithLogicC<bits<8> op, string instr_asm, SDNode OpNode,
                  InstrItinClass itin, RegisterClass RC, RegisterClass BRRegs, ↩
                      bit isComm = 0, CType type>:
  FA<op, (outs RC:$ra, BRRegs:$co), (ins RC:$rb, RC:$rc, BRRegs:$ci),
     !strconcat(instr_asm, "\t$ra, $co = $rb, $rc, $ci"),
     [], itin, type> {         // Note empty instruction matching pattern
  let shamt = 0;
  let isCommutable = 0;
  let isReMaterializable = 1;
}
```

The last step is to define the properties of the custom instruction.

```
def rvexDIVS    : ArithLogicC<0x13, "divs ", rvexDivs, IIAlu, CPURegs, BRRegs, ↩
    1, TypeIIAlu>;
```

The `rvexISelDAGToDAG` class is used to define the custom instruction selection patterns. This class implements the pattern matching code that is generated from the `tablegen` description files. The class has a separate `select` function to match instructions that have no pattern matching rules defined.

The select function uses switch statements to select between custom SDNodes. The following function implements the pattern matching rule for the `divs` instruction that replaces the extended ISD `divs` instruction with the ρ-VEX instruction `rvexDIVS`. The `rvexDIVS` instruction has been defined earlier in the `tablegen` description files.

```
  case rvexISD::Divs: {
    SDValue LHS = Node->getOperand(0);
    SDValue RHS = Node->getOperand(1);
    SDValue Cin = Node->getOperand(2);
    return CurDAG->getMachineNode(rvex::rvexDIVS, dl, MVT::i32, MVT::i32,
                                  LHS, RHS, Cin);
    break;
  }
```

3.2.4.4 Other cases

Some ρ-VEX-specific instructions, such as the `SHXADD` instructions, are easier to support and do not need custom lowering. These instructions are also defined with empty pattern matching rules so the compiler will never insert them automatically. However the `SHXADD` instruction is easier to support because we can also match an instruction to a sequence of instructions. The following code describes a rule to replace the sequence `(ADD (LHS<<1), RHS)` with `(SH1ADD LHS, RHS)`.

```
def : Pat<(add (shl CPURegs:$lhs, (i32 1)), CPURegs:$rhs),
        (SH1ADD CPURegs:$lhs, CPURegs:$rhs)>;
```

3.2.5 Floating-point operations

ρ-VEX processor does not natively support floating-point (FP) instructions. Instead software functions are used to execute FP operations. During instruction lowering FP operations are translated into library calls that will execute the instructions.

The LLVM compiler uses library functions that are compatible with the GCC Soft-FP library. The LLVM compiler-RT library is compatible with the GCC soft-FP library and is used for execution of FP instructions. Compiler-RT is a runtime library developed for LLVM that provides for these library functions.

3.2.6 Scheduling

During the scheduling pass the SDNodes are transformed into a sequential list form. Different schedulers are available for different processor types. For instance the register pressure scheduler will always try to keep the register pressure minimal which works better for x86 type processors. For the ρ-VEX processor a VLIW scheduler is used.

The list still does not contain valid assembly instructions. Virtual SSA based registers are still used and all the stack references do not reference true offsets.

3.2.7 Register allocation

During register allocation the virtual registers are mapped to available physical registers of the target processor. The register allocator considers the calling convention, reserved registers and special hardware registers during allocation. In addition the register allocator also inserts spill code when a register mapping is not available.

Liveness analysis is used to determine which virtual registers are used at a certain time. The liveness of virtual registers can be determined easily through the SSA based form of the input list. Multiple register allocation algorithms are available. All algorithms operate on the liveness information.

3.2.8 Hazard recognizer

The ρ-VEX processor has no way to recognize hazards or halt execution of code for a cycle. Because of this the correct scheduling of instructions is important for correct execution. Consider the following sequence of code:

```
  ldw $r0.2 = 0[$r0.2]   # Load from main memory
;;
  add $r0.2 = $r0.2, 2   # Add 2 to register
;;
```

After execution the register r0.2 will contain an undefined value because the load instruction has not completed execution. The compiler needs to insert an instruction or nop between the load and add instruction for correct execution. The correct instruction sequence should be the following:

```
  ldw $r0.2 = 0[$r0.2]   ## Load from main memory
;;
                         ## Empty instruction
;;
  add $r0.2 = $r0.2, 2   ## Add 2 to register
;;
```

The `ScheduleHazardRecognizer` is only able to resolve structural hazards, not data hazards. In Chapter 4 we will describe how the hazard recognizer has been combined with a new scheduling algorithm to resolve both data and structural hazards.

3.2.9 Prologue and epilogue insertion

After the register allocation pass the prologue en epilogue functions are inserted. The prologue and epilogue pass is used to calculate the correct stack offset for each variable. Code is inserted that reserves room on the stack and saves / loads variables from the stackframe.

3.2.10 VLIW Packetizer

The packetizer pass is an optional pass that is used for VLIW targets. The packetizer receives a list of sequential machine instructions that need to be bundled for VLIW processors.

The `tablegen` pipeline definition is used to build a Deterministic Finite Automata (DFA) that represents the resource usage of the processor. The DFA can be used to determine to which functional unit an instruction can be mapped and if enough functional units are available.

The DFA representation is powerful enough to consider different properties of functional units. For example consider a 4-issue VLIW processor but with only one unit supporting load / store operations. The DFA can model this pipeline and guarantee only one load / store instruction will be executed per clock cycle.

The VLIW packetizer also checks if certain instructions are legal to bundle together. The packetizer can be customized to check for hazards such as data-dependency hazards, anti-dependencies, and output-dependencies. Custom hazards can also be inserted to make sure that control flow instructions are always in a single bundle.

3.3 New LLVM features

This section describes features that have been added to the LLVM compiler. Currently, the ρ-VEX backend is the only backend that supports these kind of features.

3.3.1 Generic binary support

In [31] the design was presented for a binary format that will execute on any ρ-VEX processor, regardless of issue-width. This is achieved by generating binaries for an 8-issue ρ-VEX processor. If this binary is executed on a ρ-VEX processor with a lower issue-width the execution of the binary will be *serialized* and is performed in steps. For instance, consider executing a generic binary on a 2-issue ρ-VEX processor. 4 instruction cycles are needed to execute each part of the instruction packet.

The generic binary format adds certain instruction packet constraints. Data hazards are not allowed inside generic instruction packets because these would alter the register state of the processor. For instance, consider the instruction packet shown in Listing 3.8. A Read After Write (RAW) hazard will occur on the `r0.8` register. During the assembly phase the ordering of operations inside the instruction bundle can be altered to match the mapping of functional units in the ρ-VEX processor. When this instruction is serialized the contents of register `r0.8` is altered during the first clock cycle. During the second clock cycle this updated register will be used for execution even though the compiler intended to use the original value of `r0.8`. By disallowing RAW hazards in instruction bundles, binaries can be generated that can execute on any ρ-VEX processor irrespective of issue-width.

```
   add $r0.10   = $r0.10, $r0.11
   add $r0.12   = $r0.12, $r0.13
   add $r0.14   = $r0.14, $r0.8
   add $r0.8    = $r0.8, $r0.9
;;
```

Listing 3.8: Regular ρ-VEX binary.

To ensure that the first use of `r0.8` is executed before the last use of this register the bundle will be split as shown in Listing 3.9.

```
   add $r0.10   = $r0.10, $r0.11
   add $r0.12   = $r0.12, $r0.13
   add $r0.14   = $r0.14, $r0.8
;;
   add $r0.8    = $r0.8, $r0.9
;;
```

Listing 3.9: Generic ρ-VEX binary.

The VLIW packetization step is used to find RAW hazards between operations in instruction packets. If a RAW hazard is found the current packet is finalized and the operation is added to the next instruction packet.

3.3.2 Compiler parameterization

The HP-VEX compiler supports a machine description file that describes properties of the processor. Using this machine description certain parameters, such as issue-width, multiply units, load / store units, and branch units can be customized.

Currently the LLVM compiler supports parameterization through subtarget support. Backend features can be enabled and disabled with command line parameters. The ARM backend uses this approach to select between different ARMv7 architectures, floating-point support, and div/mul support. This approach is not be useful for the ρ-VEX processor because of the amount of features that can be customized. Each customizable feature would need a new subtarget. For example, when four features can be customized (W, X, Y and Z), then $W * X * Y * Z$ subtargets would be needed. Clearly, this approach is not usable for the ρ-VEX processor where multiple parameters can be customized with a wide range of possibilities.

A different approach is used that changes the target description during runtime of the compiler. The target processor features are described in the `rvexMCTargetDesc` class. During runtime a machine description file is read and parsed. Information from this machine description will be used to update the `rvexMCTargetDesc` class.

The following properties can be customized through the machine description file:

- **Generic binary:** disable RAW hazard check in VLIW packetizer
- **Width:** issue-width of the processor
- **Stages:** describes all the available functional units an the amount of cycles an instruction fills a functional unit.
- **Instruction itinerary:** maps an instruction itinerary to a function unit. Also describes at what cycle the output contains a legal value.

The parameters are also used to generate the DFA that will track the resource usage. During the translation of the `tablegen` file an algorithm is used to generate a DFA from the available functional units. This algorithm has been implemented in the `rvexMCTargetDesc` class.

The location of the machine description file is passed to the `rvexMCTargetDesc` through command line parameters. Custom command-line parameters can be implemented in LLVM using `cl::opt` templates.

Some parameters such as the toggling of generic binary support is not handled through the `rvexMCTargetDesc` file. These parameters are used to set global state flags that can be read at anypoint in the compilation process. The `Is_Generic_flag` is used during the `rvexVLIWPacketizer` pass.

The following code demonstrates the configuration file for a 4-issue ρ-VEX processor with support for generic binaries. The `Stages` parameter uses three values. The first value describes the amount of clock cycles an instruction occupies a functional unit. The second parameter is a bitmask for which functional unit can handle this instruction. The third parameter is currently not used.

The `InstrItinerary` parameter describes which type of instruction maps to which `Stage`. The first value references the Stage an instruction is going to use. The second parameter describes when the result of the operation is available. For example, {3, 5} describes uses the third instruction stage ({2, 2, 1}) and has a latency of 5 - 3 = 2. Each instruction itinerary that is described in `tablegen` is associated with an InstrItinerary in the configuration file.

```
Generic          = 1;
Width            = 4;
Stages           = {1, 15, 1}, {1, 1, 1}, {2, 2, 1};
InstrItinerary   = {1, 2}, {1, 2}, {3, 5}, {2, 3};
```

The compiler parameterization feature can be easily ported to other LLVM backends. The only part that has changed is the `rvexMCTargetDesc` class.

3.4 Conclusion

In this section, we have shown how the ρ-VEX backend for the LLVM compiler has been implemented. Using the `tablegen` description we have described all the instructions that are supported by the ρ-VEX processor. We have also shown how LLVM IR code is transformed into a DAG that represents the original program. Through lowering functions this DAG is transformed into a DAG that contains only operations that are supported by the target processor. During the instruction selection phase the DAG is transformed into a new DAG that contains target specific operations.

The finished DAG is transformed into a sequential list of instructions. This instruction list is used for the remaining passes. The remaining passes are used to map virtual registers to physical registers, emit prologue and epilogue functions and to perform VLIW packetization.

Certain features that are required for the ρ-VEX processor were not available in the LLVM compiler. Features such as a machine description file are new to LLVM and we have shown what changes have been made to the LLVM compiler to support machine description files. In addition, we have shown how the backend has been updated to provide support for the ρ-VEX generic binary format. Support for floating-point operations has been added by porting the LLVM floating point library to the ρ-VEX processor.

4 Optimization

The previous chapter described how the basic ρ-VEX LLVM compiler has been implemented. In this chapter we are going to discuss certain optimizations to improve the performance of binaries that are generated with the LLVM-based compiler.

4.1 Machine scheduler

A Machine Instruction scheduler is used to resolve structural and data hazards. The hazard recognizer that has been described earlier only resolves structural hazards. This means that the hazard recognizer can keep track of the functional units but it cannot keep track of data hazards between packets. The machine scheduler operates before the register allocator and is used to determine register allocation costs of each virtual register. This information is used during the register allocation pass to select a better mapping of physical to virtual registers that avoids expensive spills for commonly used virtual registers.

The Hexagon Machine scheduler has been customized to provide support for the ρ-VEX processor. The original Hexagon Machine Scheduler was used to perform packetization and to provide register allocation hints to the register allocator. The ρ-VEX machine scheduler has been enhanced with support for resolving data and structural hazards.

The machine scheduler pass uses the VLIW packetization information to build temporary instruction packets. A register allocation cost metric is used to determine an optimal scheduling and packetization of instructions. The following ρ-VEX instructions can produce data hazards that need to be resolved with the machine scheduler:

- **Multiply instructions:** 2 cycles
- **Load instructions:** 2 cycles
- **LR producing instructions:** 2 cycles
- **Compare and branch operations:** 2 cycles

The machine scheduler operates in two phases: building of an instruction queue and scheduling of the instruction queue. During initialization of the machine scheduler two queues are built: The pending instruction queue and the available instruction queue. The available queue contains all the instructions that are available to schedule before a structural hazard occurs. The scheduler will schedule instructions until the available queue is empty and checks for any remaining structural hazards. If this hazard exists, a `nop` instruction will be inserted, otherwise nothing will be done. The machine scheduler will then load new instruction from the pending queue to the available queue until a new structural hazard occurs.

The scheduling phase is used to keep track of data hazards in instruction packets. The algorithm builds temporary packets and checks for data dependencies between each packet. If a data dependency has been found a `nop` instruction is inserted to resolve the hazard. Listing 4.1 displays the algorithm in pseudo-code that is used to determine data dependencies.

```
# Get instruction from queue
Candidate = GetCandidate(Available_Queue)

if (!Candidate)
  # No Candidate found, structural hazard
  ScheduleMI(NULL, isNooped = True)

# Check latencies for all predecessors of candidate
for Predecessor in Candidate.Predecessors
  # Get instruction latency and scheduled cycle
  Latency     = Predecessor.Latency
  SchedCycle  = Predecessor.SchedCycle

  if (Latency + SchedCycle > CurrentCycle)
    # Only 1 Noop per <def> of instruction
    if (Predecessor.Nooped == false)
      # Schedule Noop
      InsertNoop = True

      #
      Predecessor.Nooped = True
      Predecessor.SchedCycle--

# Add Candidate to current packet
Reserve(Candidate)

If (Packet.full)
  # If packet is full increase scheduling cycle
  CurrentCycle++;
  Packet.clear

# Schedule instruction with Noop if required
Schedule(Candidate, InsertNoop)
```

The ρ-VEX processor has a 1 cycle delay between defining a branch register and using a branch register. The scheduler has been customized to insert `nop` instruction proceeding each branch instruction. This solution is not optimal because the empty instruction could also be used to execute an instruction that is independent. Unfortunately, this proved impossible to implement in the current version of the LLVM compiler because branch instructions are not scheduled with the machine scheduler. The machine scheduler uses branch instructions as scheduling boundaries and does not include these instructions in the temporary instruction packets. During VLIW packetization branch instructions are packed as single instructions that are not executed concurrently with other operations.

The following instruction sequence illustrates the current solution:

```
  c0   cmpgt    $b0.0, $r0.2, 9
;;

;;
  c0   br     $b0.0, .BB0_3
;;
```

The following code uses select instruction instead of branch instruction and does not need the 1 cycle delay between instructions. Select instructions are arithmetic type instructions.

```
  c0   cmpgt    $b0.0, $r0.2, 9
;;
  c0   slct     $r0.1 = $b0.0, $r0.2, $r0.3
;;
```

After the machine scheduler pass the temporary instruction bundles are deleted but the instruction ordering remains. Between the machine scheduler and VLIW packetization the only pass that can insert new code is the register allocator when spill code is introduced. During the `ExpendPredSpill` pass the spill code is checked for data dependencies and additional `nops` are inserted when required.

The final packetization occurs during the VLIW packetizer pass. The VLIW packetizer detects `nop` operations and outputs these instructions as *barrier* packets. These packets do not contain any instructions.

Scheduling could be improved further by integrating the instruction scheduling with the register allocation. The `getCandidate` function should be expanded to take into account both register pressure and resource utilization. In [32] an approach is discussed that minimizes register pressure and resource usage and can optimize performance.

4.2 Branch analysis

Analysis of assembly files generated with the LLVM-based compiler showed that branches were not handled correctly. Consider the following C code:

```
  if (c)
    return 1;
  else
    return 2;
```

This code will be roughly translated into the following assembly code:

```
  br     $b0.0, .BB0_2
;;
  goto   .BB0_1              ## Goto not required
;;
.BB0_1
  add $r0.3    = $r0.0, 1
;;
  goto   .BB0_3
;;
.BB0_2
  add $r0.3    = $r0.0, 2
;;
.BB0_3
  return
;;
```

The first `goto` operation is not required because it will jump to an adjacent block of instructions. A branch analysis pass has been developed that can recognize jumps to adjacent blocks and remove unnecessary `goto` instructions. This pass works by iterating over each `MachineBasicBlock` and finding `MachineInstr` that are branch instructions. If a branch instruction is found the instruction is analyzed to determine if the `goto` statements can be removed.

In addition, LLVM hooks for `insertbranch`, `removebranch`, and `ReverseBranchCondition` have been implemented that allow the BranchFolding pass to further optimize branches.

4.3 Generic binary optimization

Research [31] has shown that generic binaries incur a performance penalty because of inefficient register usage. Consider the previous generic binary example again:

```
  add $r0.10   = $r0.10, $r0.11
  add $r0.12   = $r0.12, $r0.13
  add $r0.14   = $r0.14, $r0.8
  add $r0.8    = $r0.8, $r0.9
;;
```

This instruction packet is not legal because of the RAW hazard that is caused by `r0.8`. The current approach to fix the RAW hazard is to split the instruction packet into two separate instructions.

```
  add $r0.10   = $r0.10, $r0.11
  add $r0.12   = $r0.12, $r0.13
  add $r0.14   = $r0.14, $r0.8
;;
  add $r0.8    = $r0.8, $r0.9
;;
```

Because more instruction packets are used than required the amount of ILP that is extracted decreases and the performance of the resulting binary decreases. Performance could be improved by using a different register for the last assignment of `r0.8`.

4.3.1 Problem statement

This kind of optimization poses a significant challenge for VLIW type compilers because the register allocation pass is executed before the VLIW packetization. This implies that the register allocator has no information about which operations will be grouped together and for which operations an extra register should be used.

A solution could be to perform VLIW packetization before register allocation is completed. This would allow the register allocation pass to determine if a RAW hazard occurs inside a packet and to assign an extra register if needed.

In practice this approach is not possible because between the register allocation pass and the VLIW packetizer pass other passes are run that can change the final code. For example, the register allocator pass can insert spill code and the prologue / epilogue insertion pass inserts code related to the stack layout. Inserting new instructions into instructions packets that have already been formed is very ugly because *packet spilling* could occur where a packet that is already full needs to move instructions to the next packet, etc.

The new machine scheduler could be customized to retain bundling information for the register allocator. Due to the complexity of implementing this approach we have chosen to change the register allocator itself and to make register allocation less aggressive.

The liveliness allocation pass determines when a virtual register is used. The register allocator uses this information to create a register mapping with minimal register pressure.

By increasing the live range of a virtual register it should be possible to force the register allocator to use more registers when multiple virtual registers are used consecutively. Consider again the previous example:

```
  add $r0.10  = $r0.10, $r0.11
  add $r0.12  = $r0.12, $r0.13
  add $r0.14  = $r0.14, $r0.8
  add $r0.8   = $r0.8, $r0.9
;;
```

This would be transformed into the following code where the final `r0.8` assignment is changed to an unused register.

```
  add $r0.10  = $r0.10, $r0.11
  add $r0.12  = $r0.12, $r0.13
  add $r0.14  = $r0.14, $r0.8
  add $r0.15  = $r0.8, $r0.9
;;
```

4.3.2 Implementation

The `LiveIntervals` pass is used to determine the live ranges of each virtual register. The pass uses the `LiveRangeCalc` class. Each virtual register has an associated `SlotIndex` which tracks when the register becomes live and when the register is killed. The `SlotIndex` class also provides method that give information on the `MachineBasicBlock` (MBB) in which the virtual register is used. The `ExtendToUses` method of the `LiveRangeCalc` class is used to update the `SlotIndex` to match the latest use of a virtual register.

Extending of the liverange has been enabled by getting the boundary of the MBB from the `SlotIndex` and by extending the `SlotIndex` to this boundary. This enables the virtual register to be live for the duration of the basic-block and will make sure the register allocator will not assign a new virtual register to a previously used physical register.

This approach is not optimal because it will increase the register usage even if RAW hazards do not occur. If more virtual registers are used then physical registers are available the execution speed will drop because extra spill code needs to be inserted.

4.4 Large immediate values

The ρ-VEX processor has support for using 32-bit immediate values. 8-bit immediate values can be handled in a single ρ-VEX operation. Values larger then 8-bit values borrow space from the adjacent ρ-VEX operations. The following code examples show the maximum amount of instruction in a packet for a 4-issue ρ-VEX processor.

```
  add $r0.10   = $r0.10, 200
  add $r0.11   = $r0.11, 200
  add $r0.12   = $r0.12, 200
  add $r0.13   = $r0.13, 200
;;
```

The following instruction packet contains an operation that uses an immediate value that cannot be contained inside a single ρ-VEX operation.

```
  add $r0.10   = $r0.10, 2000
  add $r0.11   = $r0.11, 200
  add $r0.12   = $r0.12, 200
;;
```

Large immediate values are used throughout the ρ-VEX ISA. Not only arithmetic instruction can use large immediate values but also load and store operations to represent the address offset.

```
  ldw $r0.2 = 2000[$r0.2]
;;
```

4.4.1 Problem statement

Large immediate values can be supported by creating a new instruction itinerary with support for large immediate values. This instruction itinerary would be special because it requires two functional units during VLIW packetization. The algorithm that builds the resource usage DFA needs to be updated to reflect instruction itineraries with multiple functional unit usage.

This approach is impractical for a couple of reasons. Each instruction that supports immediate values needs to be implemented twice, once for small immediate values and once for large immediate values. This would increase the risk of errors in the `Tablegen` files because each instruction has multiple definitions that need to be updated when changes are made.

A second approach is to update the VLIW packetizer to recognize large immediate values during the packetization pass.

4.4.2 Implementation

During the packetization pass each operation that uses an immediate value is checked before it is bundled in an instruction packet. If an operation is found that uses an immediate value the size of this immediate value is determined. If the immediate value is smaller then 256 then the operation is added to the instruction bundle and the DFA resource tracker is updated accordingly.

If the immediate value is larger the DFA resource tracker is used to determine if the current packet can fit the current operation plus an empty operation. If this is possible the operation is added to the instruction bundle and the DFA packetizer is updated with two new occupied resources.

4.5 Conclusion

In this chapter we discussed the optimizations that have been implemented to increase performance of ρ-VEX binaries that have been generated with the LLVM compiler.

The `rvexMachineScheduler` pass is used to handle structural and data hazards. Temporary instruction packets are generated and are filled with instructions that have been selected using cost based scheduling algorithms to reduce register pressure. The pass enables ρ-VEX binaries to execute correctly and to perform better than binaries that have not been scheduled using the `rvexMachineScheduler`.

The branch analysis optimization is used to erase unnecessary `goto` statements from the code. In addition, hooks have been provided that allow the LLVM `BranchFolding` pass to further optimize branches that are used in ρ-VEX binaries.

The generic binary optimization allows binaries with generic binary support to perform on par with regular binaries. The performance of generic binaries will only degrade once the register pressure becomes too high and spill code needs to be inserted.

The immediate value optimization allows more efficient use of available instructions of the ρ-VEX processor.

Verification and Results 5

The previous chapters described how the LLVM-based ρ-VEX compiler has been implemented. In this chapter, we are going to verify the correct operation of the compiler and we are going to measure the performance of the binaries that are generated with the LLVM-based compiler.

5.1 Simulation environment

Benchmarking and verification can be performed using the architecture simulator XSTsim and using the hardware simulator Modelsim. We have chosen to use the Modelsim hardware simulator because a complete logical simulation of the processor is used for evaluation. This provides for a more accurate simulation compared to the XSTsim simulator. Modelsim builds a simulation environment using the ρ-VEX VHDL files. A testbench is used to generate test signals. For this simulation the testbench is used to load the instruction and data memory of the ρ-VEX processor and to generate correct clock and enable signals to start execution. In Figure 5.1 the testbench process is depicted.

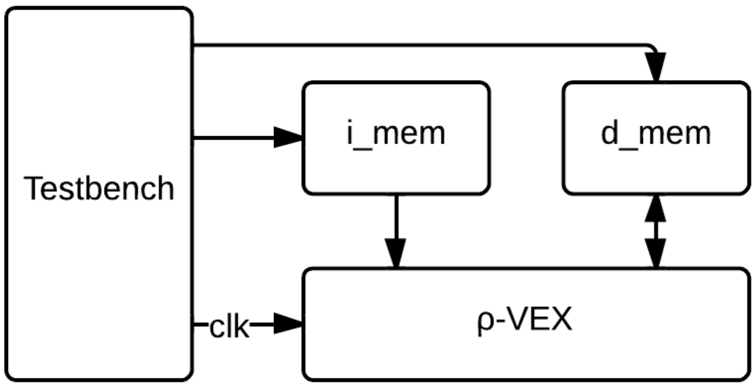

Figure 5.1: ρ-VEX testbench

Modelsim wave viewer can be used to monitor execution of the processor. The wave viewer has been used to check the contents of the registers and to monitor the instruction pipeline when necessary.

5.2 Verification

Unit testing has been used to verify the correct operation of the LLVM compiler. The tests have been performed using the XSTsim simulator and Modelsim. XSTsim can only print pipeline and register information to the terminal. It is not possible to parse strings or information from the program that is executing back to the user. To check if tests are executed correctly we check the return statement after completion of a benchmark. The value in the return register indicates whether the test executed correctly or where and at which point the test failed. Using Modelsim the contents of the return register can be monitored during execution.

- **arit.c:** Integer arithmetic tests for `char`, `short`, `int` and `long long`.
- **if.c:** Integer and boolean comparison operators.
- **float.c:** Testing of floating point library.
- **func.c:** Tests involving pointers and structures.
- **global.c:** Tests involving global integers, arrays and structures.
- **call.c:** Function calls.
- **func_pointer.c:** Function calls using function pointers.
- **loop.c:** Basic while loops.
- **misc.c:** Others tests.

During preliminary testing of the benchmark additional errors have been found. The verification tests have been updated to catch these errors. Unfortunately, some benchmarking errors are not possible to define as a unit test. Some errors, such as scheduling and register allocation errors, only occur in complex programs. Translating these errors to simple unit tests is not possible because they depend on the higher-level structure of the program.

In addition, the Powerstone benchmark [33] has been used as an additional verification step. The benchmarks consist of a number of programs that test certain functionality. In addition to performance evaluation, the benchmarks are also useful to check the executable correctness of the generated binaries.

The following benchmarks have been used for evaluation:

- **adpcm:** codec for voice compression.
- **bcnt:** Bitwise shift and operations on 1K array.
- **blit:** Graphics application.
- **compress:** UNIX compression utility.
- **crc:** Cyclic redundancy check.

- **DES:** Encryption algorithm.
- **engine:** Engine control application.
- **fir:** Finite Impulse Response filter algorithm.
- **g3fax:** Group 3 fax decode.
- **jpeg:** Image compression algorithm.
- **matrix:** Matrix multiplication.
- **pocsag:** Communication protocol for paging applications.
- **qurt:** Square root calculation using floating-point operations.
- **ucbqsort:** Quicksort algorithm.
- **v42:** Modeom encoding/decoding.

These benchmarks have been run using both Modelsim and XSTsim. A number of problems have been found during verification with the Powerstone benchmark. The following benchmarks are not able to execute properly because the benchmark appears to be broken or because the output is not verified.

- **fir:** Algorithm does not execute correctly. The algorithm also fails on a reference workstation. This indicates that the benchmark itself is faulty.
- **des:** Output of algorithm is not verified. This makes it impossible to check if the benchmark was executed correctly. This has been corrected by manually comparing the output of the des benchmark to the output of a reference workstation.

Other benchmarks produce an incorrect result when verifying using Modelsim. However, these benchmarks do produce the expected result when simulation with XSTsim.

- **jpeg:** The jpeg benchmark returns an incorrect result. Manual analysis of the benchmark does not show any errors in scheduling or instruction selection. The algorithm has been further verified by testing subsets of the jpeg algorithm. All the individual subsets work as expected and produce the expected results. The jpeg benchmarks works by executing these subsets 600 times. The code is demonstrated in Listing 5.1.

```
for (i = 0; i < 600; i++)
    huff_dc_dec(&Data)

for (i = 0; i < 600; i++)
    huff_ac_dec(&Data);

for (i = 0; i < 600; i++)
    dquantz_lum(&Data);

for (i = 0; i < 600; i++)
    j_rev_dct(&Data);
```

Listing 5.1: `jpeg` code example

Benchmark	**W2**	**W4**	**W8**
`adpcm`	2.206.300	2.115.820	2.118.040
`bcnt`	39.540	38.200	35.640
`blit`	1.164.280	1.204.140	1.204.120
`compress`	4.034.600	3.886.340	3.872.920
`crc`	1.323.160	1.290.540	1.290.560
`des`	2.755.280	2.423.420	2.186.540
`engine`	22.820.980	18.119.680	18.119.680
`g3fax`	55.537.540	56.645.740	56.680.260
`matrix`	717.100	615.060	615.040
`pocsag`	2.453.040	1.987.140	1.954.860
`qurt`	1.378.430	1.331.560	1.331.660
`ucbqsort`	9.568.680	9.457.160	9.454.600
`v42`	106.505.820	103.197.300	97.380.520

Table 5.1: LLVM-based compiler performance in ns

When the `for` loop range is changed to a smaller number, such as 500, the benchmark produces the expected result. This indicates a possible simulation error where a part of the data memory is overwritten during execution.

Because these benchmark are not running correctly we will not consider them for measuring the performance of the LLVM-based compiler.

5.3 Benchmark results

The Powerstone benchmark has been used to evaluate the performance of the LLVM-based compiler. We measure the execution time of the generated binaries and also the compile-time for each binary.

5.3.1 General performance

General performance of all the compilers that target the ρ-VEX processor are shown in Tables 5.1, 5.2 and 5.3. The absolute performance of the compilers is displayed in Figure 5.2.

5.3.1.1 HP-VEX performance

In Table 5.2 the absolute performance of the HP-VEX compiler is given. As expected the HP-VEX compiler generates the best performing binaries. Table 5.4 shows that the HP-based compiler binaries are on average twice as fast as the LLVM-based compiler binaries. The HP-VEX compiler is unable to generate working binaries for the `adpcm` and `g3fax` benchmark when targeting a 8-issue ρ-VEX processor. The reason for this error is not immediately clear.

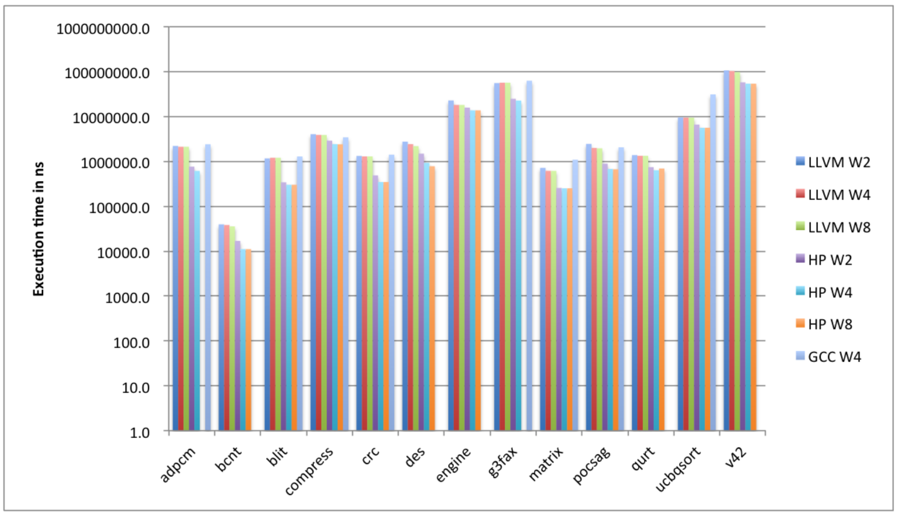

Figure 5.2: Absolute performance

The HP-VEX compiler generates excellent performing binaries because it has a superiour scheduling techniques for finding and extracting ILP in source code. The HP-VEX compiler uses a trace-based scheduling technique that enables better ILP extraction. In addition to this, the HP-VEX compiler also seems to do more optimization. Even when compiling with `-O0` the compiler already performs certain optimizations that are not available for LLVM-based compiler. For example, compare the output for the following simple C program:

```
int main() {
    int a = 3, b = 2, c;

    c = a + b;

    return c;
}
```

Listing 5.2 displays the output of the HP-VEX compiler and Listing 5.3 shows the output of the LLVM-based compiler. The output shows immediatly that the HP-VEX compiler has eliminated the add operation and has copied the final value of the operation straight to the return register. The LLVM compiler does not perform these kinds of optimizations at `-O0`.

```

    add $r0.3    = $r0.0, 5       ## Move to return register
;;
    return
;;
```

Listing 5.2: HP compiler output

Benchmark	W2	W4	W8	Comment
adpcm	763.120	613.520	ERR	Infinite loop
bcnt	16.860	11.100	11.080	
blit	341.620	301.360	301.360	
compress	2.893.080	2.421.420	2.400.140	
crc	486.980	347.040	347.040	
des	1.485.980	930.360	779.980	
engine	15.785.220	13.720.900	13.719.860	
g3fax	24.821.120	22.678.220	ERR	Infinite loop
matrix	255.940	251.260	251.220	
pocsag	889.160	676.680	667.880	
Quicksort	743.480	633.520	692.840	
ucbqsort	6.621.460	5.604.720	5.603.460	
v42	57.672.780	53.910.400	53.910.260	

Table 5.2: HP-based compiler performance in ns

```
    add $r0.2   = $r0.0, 2
    add $r0.3   = $r0.0, 3
;;
    add $r0.3   = $r0.2, $r0.3   ## Move to return register
;;
    return
;;
```

Listing 5.3: LLVM compiler output

Figure 5.3 shows the relative performance of LLVM-based binaries compared to HP-VEX binaries. As expected HP-VEX binaries perform better then LLVM-based binaries. Inspection of the generated assembly files indicates that the HP-VEX compiler is able to fill significantly more functional units than the LLVM-based compiler. Further the LLVM-based compiler is overly aggressive in inserting `nop` instructions to reduce structural and data hazards.

5.3.1.2 GCC performance

Table 5.3 shows the absolute performance of the GCC-based compiler. Figure 5.4 shows the relative performance of LLVM-based binaries compared to GCC-based binaries. The GCC based compiler is unable to compile a significant number of Powerstone benchmarks.

Performance of the LLVM-based compiler shows mixed results. Some benchmarks perform significantly better such as `matrix` and `ucbqsort` but both `compress` perform worse. Manual inspection of this benchmark does not show an obvious reason for decreased performance except for the overly aggressive `nop` insertion of the LLVM machine scheduler.

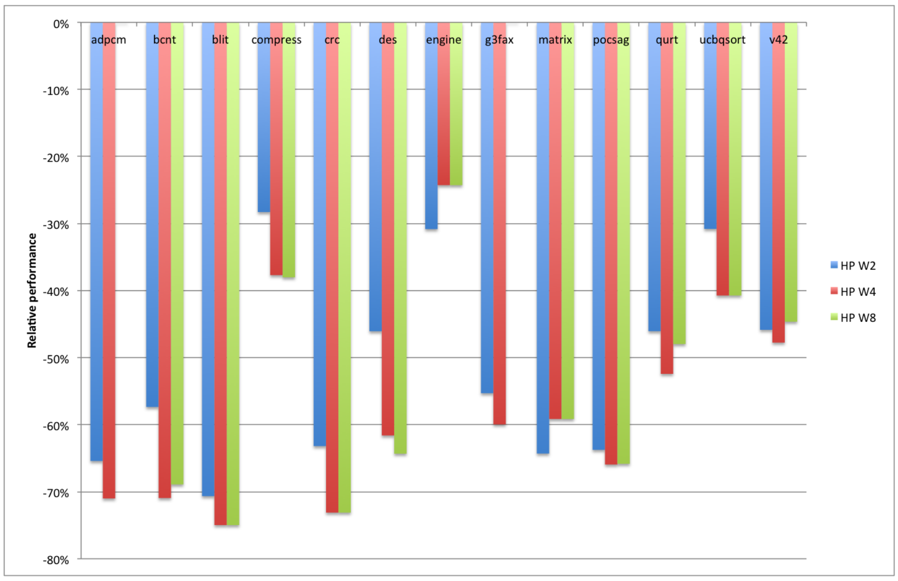

Figure 5.3: HP-LLVM relative performance

Benchmark	W4	Comment
`adpcm`	2.404.760	Wrong result
`bcnt`	ERR	Infinite loop
`blit`	1.284.340	Wrong result
`compress`	3.433.960	
`crc`	1.409.960	
`des`	ERR	Error load
`engine`	ERR	Error load
`g3fax`	62.666.300	
`matrix`	1.090.420	
`pocsag`	2.058.700	
`qurt`	ERR	
`ucbqsort`	31.075.420	
`v42`	ERR	

Table 5.3: GCC-based compiler performance in ns

5.3.1.3 Issue-width

Close inspection of the absolute performance reveals some interesting facts related to the issue-width of the processor. Increasing the issue-width from 2-issue to 4-issue leads to a significant increase in performance. Further increasing the issue-width to 8-issue machine does not lead to an increase in performance. This is shown in Figure 5.5. For some benchmarks such as `bcnt` the LLVM-based compiler is able to find extra parallelism but for most benchmarks the performance increase is non-existent.

Manual inspection of the assembly files that are generated shows that both compilers

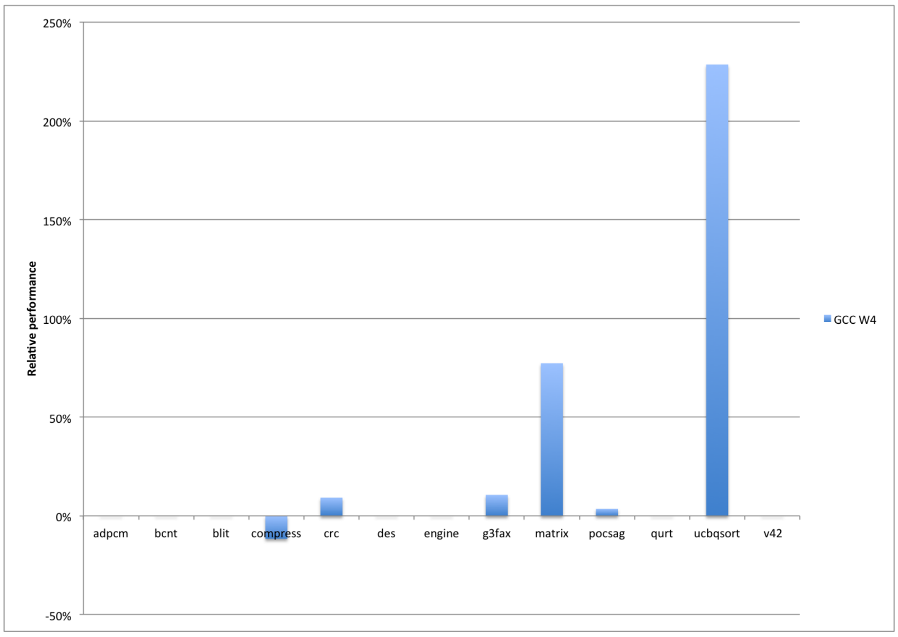

Figure 5.4: GCC-LLVM relative performance

Benchmark	VEX W2	VEX W4	VEX W8	GCC W4
adpcm	-65%	-71%	ERR	ERR
bcnt	-57%	-71%	-69%	ERR
blit	-71%	-75%	-75%	ERR
compress	-28%	-38%	-38%	-12%
crc	-63%	-73%	-73%	7%
des	-46%	-62%	-64%	ERR
engine	-31%	-24%	-24%	ERR
g3fax	-55%	-60%	ERR	13%
matrix	-64%	-59%	-59%	52%
pocsag	-64%	-66%	-66%	-16%
qurt	-46%	-52%	-48%	ERR
ucbqsort	-31%	-41%	-41%	225%
v42	-46%	-48%	-45%	ERR

Table 5.4: Relative performance of LLVM-compiler binaries

are able to generate instruction packets that use a higher number of functional units. These large instruction packets do not lead to an increase in performance because they are not contained inside loop structures. The amount of times large instruction packets are executed is limited.

Some benchmarks, such as `blit` and `pocsag`, perform worse when using a higher issue-width. Analysis shows that the machine scheduler adds unnecessary `nop` instructions that cause a decrease in performance.

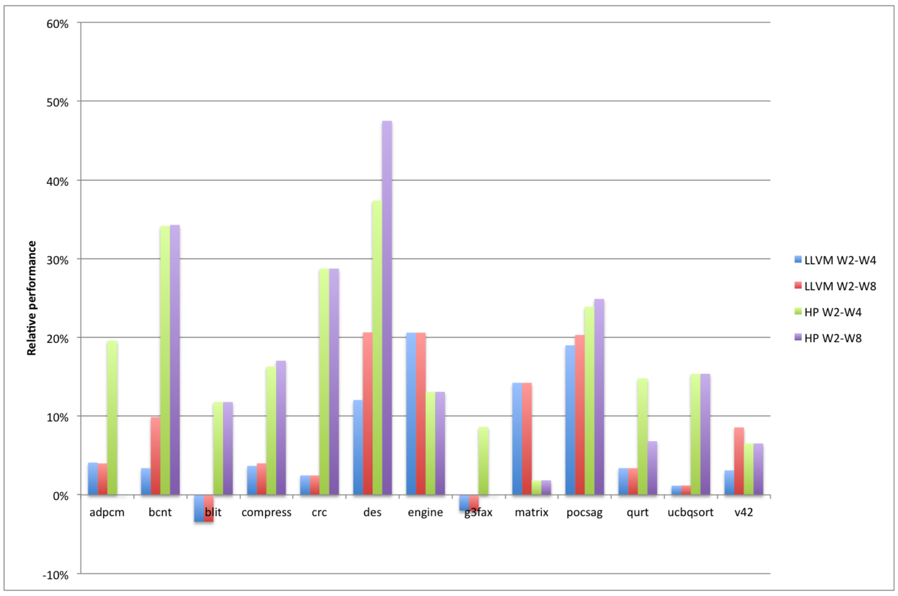

Figure 5.5: Relative performance for increasing issue-width

5.3.1.4 Compiler optimizations

Unfortunately it was not possible to compare the compilers for different optimization levels. The LLVM-based compiler needs a *runtime* library for most benchmarks when compiling with optimizations turned on. This requires us to port the complete LLVM-based *runtime* library to the ρ-VEX processor. For this thesis only the floating-point library has been ported. Porting the complete *runtime* library falls outside the scope of this thesis.

5.3.2 Generic binary

The performance of generic binaries has been tested by generating three sets of binaries for a 4-issue ρ-VEX processor: Regular binary, Generic binary without optimizations and Generic binary with optimizations. These simulations have been performed with the XSTsim architectural simulator. The relative performance of generic binaries compared to regular binaries is displayed in Figure 5.6.

Figure 5.6 shows that the optimization for generic binaries provides for more efficient generic binaries for most benchmarks. The `matrix` benchmark in particular shows excellent increase in performance. The `adpcm` and `bcnt` benchmark however shows a significant decrease in performance. Closer inspection of the generated binaries shows that some benchmarks start spending a lot of time executing spill code to save registers to the stack. This is related to the optimization being too aggressive for `MachineBasicBlocks` with a lot of virtual registers. The optimization causes virtual registers to remain live from the moment they are defined to the end of the `MachineBasicBlock`. The optimization should be optimized to only keep the virtual register live for the duration of the next instruction packet. This would reduce virtual register usage and would also reduce

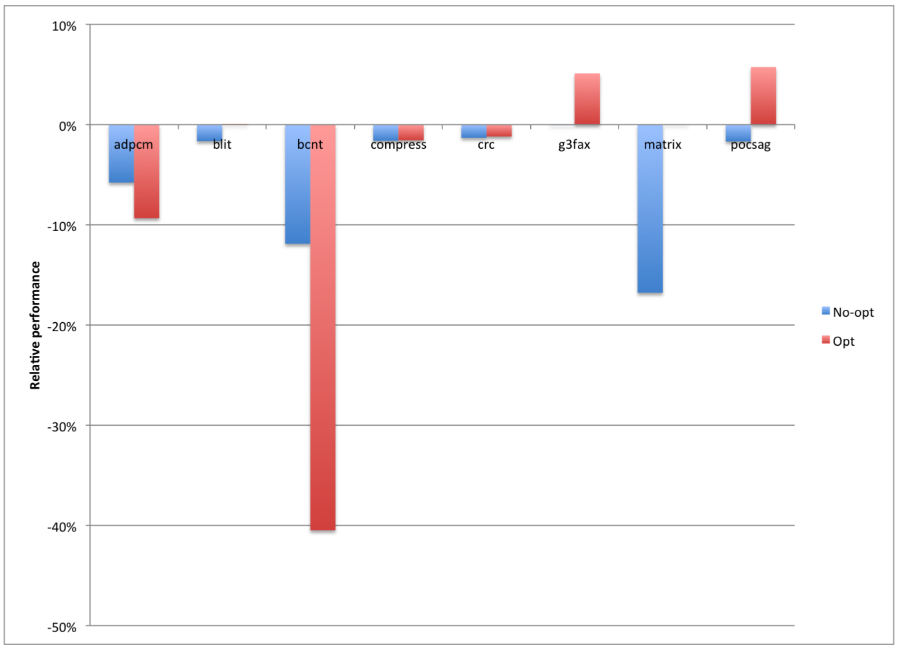

Figure 5.6: Generic-Regular performance

Benchmark	Regular	No opt	Difference	With opt	Difference
adpcm	176.327	187.135	-6%	194.492	-9%
blit	100.328	102.007	-2%	100.325	0%
bcnt	3.358	3.811	-12%	5.643	-40%
compress	385.797	392.115	-2%	391.920	-2%
crc	107.528	108.950	-1%	108.820	-1%
g3fax	5.124.268	5.124.433	0%	4.874.923	5%
matrix	51.238	61.572	-17%	51.238	0%
pocsag	174.128	177.103	-2%	164.661	6%

Table 5.5: Generic binary performance

spill code that is introduced.

Some benchmarks show an increase in performance when compiling using the generic binary optimization. This shows how aggressive the register allocator can be in trying to reduce register pressure. Even though the ρ-VEX processor has 64 general-purpose registers available the compiler will always try to keep the register usage down to a minimum.

[31] stated that performance should increase if more registers are used for generic binaries. The amount of different registers that are used by each benchmark have been tracked and are displayed in Table 5.6. The table shows that the optimization is able to increase the register usage for each benchmark. The `adpcm` benchmark already uses a large amount of registers and it is not possible to increase this by much. This probably causes the excessive spill code that is generated for this benchmark.

Benchmark	Regular	No-opt	Opt
`adpcm`	49	49	56
`blit`	18	18	33
`bcnt`	14	14	59
`compress`	27	27	60
`crc`	19	19	27
`g3fax`	18	18	19
`matrix`	18	18	36
`pocsag`	24	24	31

Table 5.6: Register usage

More problematic is the fact that the optimization breaks certain benchmarks. Table 5.5 only shows the benchmarks that were able to execute properly. The large number of virtual registers that are used in certain benchmarks are causing unforeseen problems during the compilation process. For instance, during compilation of the `compress` benchmark the register allocator tried to use the `PC` register as a physical register. This was quickly fixed but indicates the kind of problems that are introduced.

5.3.3 Compile-time

The compile time has been measured on a virtual machine running *Ubuntu 12.04*. The virtual machine has one processor and 1024MB RAM allocated. The host system uses a 1.7GHz Intel Core i5 processor. The compile-time has been measured by compiling each benchmark 100 times. The Linux `time` command has been used to measure the execution time.

The HP-VEX and GCC compiler are able to generate assembly files from the input C source code. The LLVM compiler uses a two-phase compilation where Clang is used to compile to LLVM IR and the LLVM static compiler (`LLC`) is used to compile to assembler. This two-phase compilation will have a negative effect on the performance because extra files need to be read and written from the main memory. This could be avoided by passing the output of Clang straight to `LLC` but this was not possible to achieve in a timely manner.

The compile-time of each benchmark is given in Table 5.7 and depicted in Figure 5.7. The GCC-based compiler outperforms both the HP-VEX and LLVM-based compiler in nearly every benchmark. The HP-VEX compiler offers no timing report, which makes it impossible to determine which part of compilation is causing the delay. The lower performance could be related to the extra optimization that the HP-VEX compiler performs.

The LLVM-based compiler is slower in nearly every benchmark compared to the GCC-based compiler. This could be related to the extra time that is required to write the LLVM IR files to the disk.

Benchmark	HP VEX	GCC	LLVM
adpcm	6,41	6,55	5,68
bcnt	1,72	2,02	2,91
blit	2,26	2,12	3,72
compress	8,82	7,19	
crc	2,54	2,30	3,28
engine	5,60	3,64	4,14
g3fax	5,00	3,95	6,58
matrix	1,97	1,88	2,92
pocsag	6,60	4,28	5,49
qurt	2,74	3,46	3,33
ucbqsort	8,41	3,60	4,54

Table 5.7: Compile-time in seconds

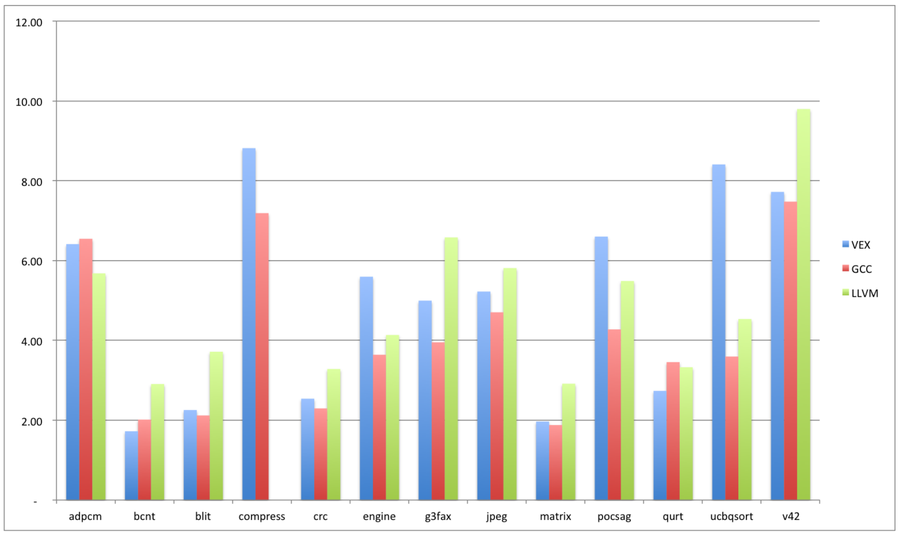

Figure 5.7: Compile-time in seconds

5.4 Conclusion

In this section we have shown how the operation of the LLVM-based compiler has been verified and how well binaries execute that have been generated with the LLVM-based compiler.

The benchmarks and verifications have shown that the LLVM-based compiler still contains bugs. Not all benchmarks are able to execute using the Modelsim simulator but all benchmarks are able to execute using the XSTsim simulator. The reason for the failing benchmarks remain unclear. It is possible that the compiler generates code that is not scheduled properly but analysis of the `jpeg` benchmark indicated the possibility of other issues.

We have shown that the LLVM-based compiler exceeds the performance of the GCC-based compiler but the compiler is still outperformed by the HP-VEX compiler. As expected the HP-VEX compiler generates binaries that perform very well. This is related to the trace-based scheduling techniques that are employed to extract a high level of ILP [15]. In addition, the HP-VEX compiler also performs certain optimizations that are not available to the LLVM-based compiler at `-O0`.

Additionally, the benchmarks have also shown that the LLVM-based compiler is the only compiler able to generate correct code for all selected benchmarks. Surprisingly, even the HP-VEX compiler generates incorrect binaries for certain benchmarks. The code quality of the GCC-based compiler is bad with four benchmarks failing to execute.

Furthermore, we have shown that the generic binary optimization allows generic binaries to operate at speeds that are nearly equal to the regular binaries. The generic binary optimization does introduce spill code in benchmarks that already use a large number of physical registers. The current optimization is too aggressive for the `adpcm` benchmark and introduces excessive amount of spill code that degrades performance. The optimization should be fine tuned to consider this situation. More problematic is the fact that the used optimization breaks certain benchmarks.

Conclusion 6

The previous chapters discussed how the LLVM-based backend has been implemented. This chapter summarizes the results and presents opportunities for future research.

6.1 Summary

In chapter 2 we discussed the background of the ρ-VEX processor and the basics of the LLVM compiler framework. The ρ-VEX processor is a VLIW type processor that uses RISC like instructions to operate. We have presented the basic design of the processor, the instructions that are supported, register properties, and the run-time architecture has been discussed. This information will be used during implementation of the LLVM-based compiler.

We have also discussed the basic working of the LLVM compiler framework. Building a ρ-VEX backend for the LLVM compiler is feasible because the current version of the LLVM compiler already targets a VLIW processor.

Finally we have also discussed how the LLVM compiler will be verified. The verification step is extremely important to determine whether the binaries that are generated work correctly.

Chapter 3 discussed how the LLVM-based compiler has been implemented. Using the `tablegen` description we have described all the instructions that are supported by the ρ-VEX processor. We have also shown how LLVM IR code is transformed into a DAG that represents the original program. Through lowering functions this DAG is transformed into a DAG that contains only operations that are supported by the target processor. During the instruction selection phase the DAG is transformed into a new DAG that contains target specific operations.

The finished DAG is transformed into a sequential list of instructions. This instruction list is used for the remaining passes. The remaining passes are used to map virtual registers to physical registers, emit prologue and epilogue functions and to perform VLIW packetization.

Certain features that are required for the ρ-VEX processor are not available in the LLVM compiler. Features such as a machine description file are new to LLVM and we have shown what changes have been made to the LLVM compiler to support machine description files. In addition, we have shown how the backend has been updated to provide support for the ρ-VEX generic binary format. Support for floating point operations has been added by porting the LLVM floating point library to the ρ-VEX processor.

Chapter 4 discussed the optimizations that have been implemented to improve performance of the ρ-VEX binaries. The `rvexMachineScheduler` pass is used to handle structural and data hazards. Temporary instruction packets are generated and are filled with instructions that have been selected using cost based scheduling algorithms to reduce register pressure. The pass enables ρ-VEX binaries to execute correclty and to perform better than binaries that have not been scheduled using the `rvexMachineScheduler`.

The branch analysis optimization is used to erase unnecessary `goto` statements from the code. In addition, hooks have been provided that allow the LLVM `BranchFolding` pass to further optimize branches that are used in ρ-VEX binaries.

The generic binary optimization allows binaries with generic binary support to perform on par with regular binaries. The performance of generic binaries will only degrade once the register pressure becomes too high and spill code needs to be inserted.

The immediate value optimization allows more efficient use of available instructions of the ρ-VEX processor.

Finally, in chapter 5 we have shown how the operation of the LLVM-based compiler has been verified and how well binaries execute that have been generated with the LLVM-based compiler.

The benchmarks and verifications have shown that the LLVM-based compiler still contains bugs. Not all benchmarks are able to execute using the Modelsim simulator but all benchmarks are able to execute using the XSTsim simulator. The exact reason for the failing benchmarks remain unclear. It is also possible that the compiler generates code that is not scheduled properly but analysis of the `jpeg` benchmark indicated the possibility of other issues.

We have shown that the LLVM-based compiler exceeds the performance of the GCC-based compiler but the compiler is still outperformed by the HP-VEX compiler. As expected the HP-VEX compiler generates binaries that perform very well. This is related to the trace based scheduling techniques that are employed to extract a high level of ILP [15]. In addition, the HP-VEX compiler also performs certain optimizations that are not available to the LLVM-based compiler at `-O0`.

Additionally, the benchmarks have also shown that the LLVM-based compiler is the only compiler able to generate correct code for all selected benchmarks. Surprisingly, even the HP-VEX compiler generates incorrect binaries for certain benchmarks. The code quality of the GCC-based compiler is bad with four benchmarks failing to execute.

We have also shown that the generic binary optimization allows generic binaries to operate at speeds that are nearly equal to the regular binaries. The generic binary optimization does introduce spill code in benchmarks that already use a large number of physical registers. The current optimization is too aggressive for the `adpcm` benchmark and introduces excessive amount of spill code that degrades performance. The optimization should be fine tuned to consider this situation. More problematic is the fact that the used optimization breaks certain benchmarks.

6.2 Main contributions

In this thesis we have presented the design of a LLVM-based ρ-VEX compiler. We have shown how the compiler is built, what optimizations have been implemented and the performance of binaries that have been generated with the LLVM-based compiler.

The following parts have been contributed to the ρ-VEX project:

- LLVM-based ρ-VEX compiler.

  - Floating point support.
  - 64-bit integer support

- Parameterization of LLVM-based compilers.
- Support for generic binaries.
- Scheduling optimizations that improve performance of regular binaries.
- Register allocation optimization that improves performance of generic binaries

6.3 Future work

Future areas of research for the LLVM-based ρ-VEX could involve the following subjects.

- **Fix compilation bugs:** The current version of the LLVM-based compiler shows possible erors in a number of benchmarks. These bugs need to be fixed to produce a more stable version of the LLVM-based compiler
- **Optimization support:** Currently we have not considered using higher optimization levels. Brief tests with higher optimization levels indicated massive increases in performance. Unfortunately, the compiler started using runtime library functions that have not yet been ported to the ρ-VEX processor.
- **Optimize scheduling:** The scheduling algorithms that are used by the LLVM-based compiler can be further optimized. In [34] a technique is presented to optimize the scheduling for VLIW processors. The algorithm uses information about the instruction delay to reorder the instructions. This leads to an increase in pipeline utilization and code that performs better. Scheduling could also be improved by integrating the register allocation and instruction scheduling phase as discussed in [32].
- **Enhance parameterization:** At the moment only issue-width, instruction stages and instruction delay can be customized through configuration files. In the future more configuration options can be added such as number of registers available or scheduling parameters.
- **LLVM JIT:** The LLVM compiler support Just-In-Time compilation through the LLVM interpreter. Implementing the interpreter for the ρ-VEX processor could produce interesting results where binary properties can be modified during runtime. For example the interpreter could look for code with higher degree of ILP. If suitable code is found the program will be executed on an 8-issue ρ-VEX processor. If suitable code is not found the issue-width could be reduced to 2- or 4-issue code and the idle functional units can be shut down to preserve energy.
- **Trace based scheduling:** Currently the LLVM compiler uses a basic block scheduler. Introducing a trace-based scheduler could further improve performance of binaries for VLIW type processors.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design.* Morgan Kaufmann, 2009.

[2] R. Seedorf, F. Anjam, A. Brandon, and S. Wong, "Design of a pipelined and parameterized vliw processor: -vex v2.0," *6th HiPEAC Workshop on Reconfigurable Computing (WRC 2012)*, 2012.

[3] J. A. Fisher, *The VEX System*, pdf.

[4] T. van As, "ρ-vex: A reconfigurable and extensible vliw processor-vex: A reconfigurable and extensible vliw processor," Master's thesis, Technical university Delft, Mekelweg 4, 2628 CD, Delft, The Netherlands, 2008.

[5] P. Faraboschi, G. Brown, J. Fisher, G. Desoll, and F. Homewood, "Lx: a technology platform for customizable vliw embedded processing," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, 2000, pp. 203–213.

[6] J. A. Fisher, "Very long instruction word architectures and the eli-512," *SIGARCH Comput. Archit. News*, vol. 11, no. 3, pp. 140–150, Jun. 1983. [Online]. Available: http://dl.acm.org/citation.cfm?id=1067651.801649

[7] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, Jan 1967.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal Q1*, 2001.

[9] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach, Fifth edition.* Morgan Kaufmann, 2012, ch. Chapter Three: Intstruction-Level Parallelism and Its Exploitation, p. 244.

[10] N. Sakran, M. Yuffe, M. Mehalel, J. Doweck, E. Knoll, and A. Kovacs, "The implementation of the 65nm dual-core 64b merom processor," in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, Feb 2007, pp. 106–590.

[11] D. W. Wall, "Limits of instruction-level parallelism," *WRL Research Report*, p. 73, November 1993.

[12] M. Lam, "Software pipelining: An effective scheduling technique for vliw machines," *SIGPLAN Not.*, vol. 23, no. 7, pp. 318–328, Jun. 1988. [Online]. Available: http://doi.acm.org/10.1145/960116.54022

[13] B. D. de Dinechin, "From machine scheduling to vliw instruction scheduling," 2004.

[14] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman, "A vliw architecture for a trace scheduling compiler," *Computers, IEEE Transactions on*, vol. 37, no. 8, pp. 967–979, Aug 1988.

[15] P. G. Lowney, P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'donnell, and J. C. Ruttenberg, "The multiflow trace scheduling compiler," *JOURNAL OF SUPERCOMPUTING*, vol. 7, pp. 51–142, 1993.

[16] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: A VLIW Approach to Architecture, Compilers and Tools.* Elsevier, 2005.

[17] *VEX Toolchain*, HP Inc. [Online]. Available: http://www.hpl.hp.com/downloads/vex/

[18] *GCC, the GNU Compiler Collection*, Free Software Foundation, Inc. [Online]. Available: http://gcc.gnu.org

[19] V. A. Chris Lattner, "Llvm: A compilation framework for lifelong program analysis & transformation," *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[20] L. Codrescu, E. Plondke, W. Anderson, C. Maule, C. Koob, M. Zeng, A. Ingle, C. Tabony, and suresh venkumahanti, "Qualcomm hexagon dsp: An architecture optimized for mobile multimedia and communications," *IEEE Micro*, vol. 99, no. 1, p. 1, 5555.

[21] L. T. Simpson, "Porting llvm to a next generation dsp," *LLVM Developers' Meeting*, 2011.

[22] C. Erhardt, "Design and implementation of a tricore backend for the llvm compiler framework," Master's thesis, Friedrich-Alexander-Universita t Erlangen-Nürnberg, 2009.

[23] L. Antani, H. Ansari, and A. Parameswaran, "Tricore port for gcc - an analysis," Master's thesis, Indian Institute of Technology, Bombay Mumbai, Unknown.

[24] R. Seedorf, "Fingerprint verification on the vex processor," Master's thesis, Technical university Delft, 2011.

[25] LLVM, "Auto-vectorization in llvm." [Online]. Available: http://llvm.org/docs/Vectorizers.html

[26] ——, "Dragonegg - using llvm as a gcc backend." [Online]. Available: http://dragonegg.llvm.org

[27] Clang. Clang - features and goals. [Online]. Available: http://clang.llvm.org/features.html

[28] C. Lattner, "The llvm compiler framework and infrastructure (part 1)," Presentation.

[29] J. L. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, and T. Gross, “Design of a high performance vlsi processor,” Stanford, CA, USA, Tech. Rep., 1983.

[30] R. Garner, A. Agrawal, F. Briggs, E. Brown, D. Hough, B. Joy, S. Kleiman, S. Muchnick, M. Namjoo, D. Patterson, J. Pendleton, and R. Tuck, “The scalable processor architecture (sparc),” in *Compcon Spring ’88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, Feb 1988, pp. 278–283.

[31] A. Brandon and S. Wong, “Support for dynamic issue width in vliw processors using generic binaries,” *EDAA*, 2013.

[32] D. G. Bradlee, S. J. Eggers, and R. R. Henry, “Integrating register allocation and instruction scheduling for riscs,” *SIGPLAN Not.*, vol. 26, no. 4, pp. 122–131, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/106973.106986

[33] J. Scott, L. H. Lee, J. Arends, and B. Moyer, “Designing the low-power m*core architecture,” in *Proc. IEEE Power Driven Microarchitecture Workshop*, 1998.

[34] M. Vahedi, “Iterative instruction scheduling for a vliw processor,” Master’s thesis, Technical University Delft, 2013.

LLVM Quickstart guide A

In this section we will describe how the LLVM-based compiler can be used for projects. This guide assumes that *clang* and*vex_llv* executables are available. Further we also require the availability of the VEX assembler and VEX linker.

All the source code that is required for this guide is contained in the `source.zip` archive. This archive contains the following folders:

- **config:** Configuration files for vex_llc. Different configuration files exist for 2-issue, 4-issue, and 8-issue machines.
- **benchmarks:** Folder containing the powerstone benchmark suite.
- **tests:** Folder containing the LLVM verification tests.
- **float:** Folder containing the floating point library used during compilation.

Each folder contains a `Makefile` that automates the build process. The current version of the `Makefiles` are used to compile `c` sourcecode into ρ-VEX assembler.

A.1 Compilation

The following code uses *clang* to compile input `c` sourcecode into LLVM IR code.

```
clang -emit-llvm -S -m32 -O0 -fno-stack-protector input.c -o output.ll
```

The compiler flags are used as follows:

- `-emit-llvm`: Emits the LLVM IR code.
- `-s`: Emit human readable assembler.
- `-m32`: `int` is 32-bits.
- `-O0`: No optimizations.
- `-fno-stack-protector`: Do not emit a stack-protector. The current run-time library does not support stackprotection.

vex_llc is used as follows:

```
vex_llc input.ll -march=rvex -mcpu=rvex-vliw -config=../config/rvex_W4_2 -↩
    enable-misched=true -relocation-model=static -o output.s
```

The compiler flags are used as follows:

- `-march`: Select the architecture to compile for.
- `-mcpu`: Select the subtarget of the architecture.
- `-config`: Path to configuration file.
- `-enable-misched`: Enables the machine scheduler that is required for correct scheduling of ρ-VEX operations.
- `-relocation-model`: Current version of ρ-VEX processor has no support for dynamic binaries.

A executable can be generated using *rvex-as* and *rvex-ld* as follows:

```
rvex-as --issue 4 --borrow 1,3.0,2.3,1.2,0. --config 9335 -h input.s -o ↩
    output.o
rvex-ld  output.o _start.o -o output
```

The assembler flags are used as follows:

- `-issue`: Select issue width of final binary.
- `-borrow`: Borrowing scheme used for large immediate values.
- `-config`: Describes which functional units support Multiply, Branch and Load instructions.
- `-h`: Flag toggles generic binaries.

The linker is used to build the final executable using a startup file and all the object files.

A.2 Simulation

Simulation can be performed using XSTsim or with Modelsim. XSTsim can

```
xstsim-r-VEX-1.1.3 --ips='"[r-VEX c]"'  --c.trace=5 --c.trace_regs=2 --↩
    accuracy=0 --c.target_exec='"test"'
```

The flags that are used for xstsim are described in the xstsim documentation.

For simulation with modelsim `hex` files are required. These can be generated with the *rvex-elf2vhd* tool as follows:

```
rvex-elf2vhd --hex test
```

The *elf2vhd* tool can be used to generate `hex` and `vhd` files with the following flags.

- `-hex`: Generates `hex` files.

- `-vhd:` Generates `vhd` files.

The `vhd` can only be used for 4-issue width ρ-VEX processors because the instruction memory datatype is fixed to 128-bits. The `hex` files can be used for all issue widths an can also be used to load the instruction and data memory of physical hardware.

B LLVM Development guide

This section describes how to build LLVM from source. The following software is required:

- **Compiler:** A compiler is obviously required to build LLVM. On Linux LLVM can be compiled with LLVM and GCC. Mac users can build LLVM using LLVM.
- **CMAKE:** CMAKE is required to generate the build files for LLVM.
- **git:** Required to get the source files

The source files can be downloaded from the following repository:

```
git@github.com:zerokill/lbd.git
```

B.1 Building LLVM from source

CMAKE is used to generate the build scripts for LLVM. On Mac systems the following command is used to generate a *xcode* project that can build LLVM:

```
mkdir build
cd build
cmake -G "Xcode" ../lbd/ -DLLVM_TARGETS_TO_BUILD="rvex"
```

- `../lbd/`: Points to the LLVM source directory
- `-DLLVM_TARGETS_TO_BUILD`: Selects the target architectures that should be supported. Multiple architectures are possible.

LLVM can be build using the following command:

```
make vex\_llc
```