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

CE-MS-2014

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 Utrecht, 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

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2014

Committee Members :

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

VLIW	Very Long Instruction Word
ILP	Instruction Level Parallelism
OoO	Out-of-Order Execution
CPI	Clocks Per Instruction
LLVM	Low Level Virtual Machine
GCC	GNU Compiler Collection
ISD	Instruction SelectionDAG
DAG	Directed Acyclic Graph
DFA	Deterministic Finite Automaton

Acknowledgements

Maurice Daverveldt
Delft, The Netherlands
March 1, 2014

Introduction 1

¡INTRODUCTIE TEKST¿

1.1 Motivation

In 2008 Thijs van As designed the first version of the ρ-VEX processor [3]. 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 [4] from HP/STMicroelectronics. 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. A GCC based VLIW compiler has been developed by IBM. A Very Long Instruction Word (VLIW) processor 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.

A regular RISC type processor, such as the MIPS and ARM processor, contain 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. A pipeline can contain multiple instructions that are in different stages of execution.

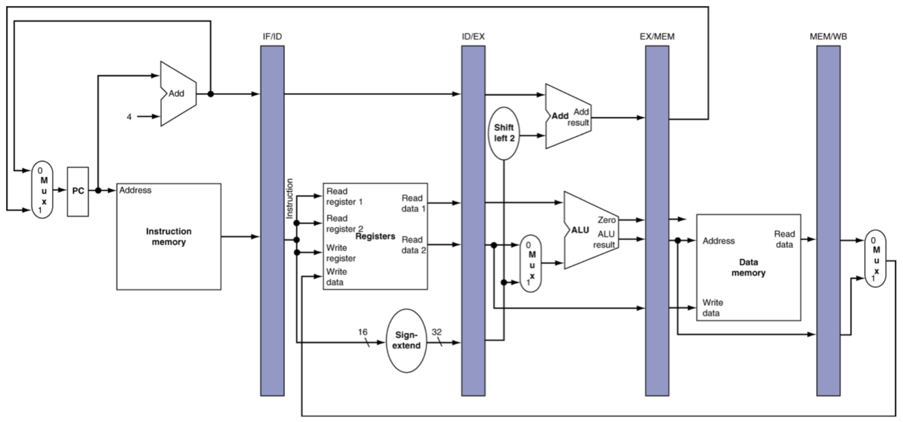

Figure 1.1: MIPS pipeline [1]

Introducing pipelining will generally lower the Clocks Per Instruction (CPI) rate of a processor to below 1.0. 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.

If a higher then 1.0 CPI is desired multiple instructions need to be executed during a single clock cycle. Machines that can execute multiple instructions are called multi issue 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 Tomasulos algorithm 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 demonstrated the limitations of further ILP extraction in a spectacular way. It used a 20-stage pipeline [5] 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 expected 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.

[6] showed the actual limitations of ILP extraction in hardware and demonstrated that other techniques need to be used to find and execute more ILP.

VLIW differs from multiple issue machines in that parallelism is found during compile-time instead of during run-time. 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 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 [7]. Multiflow designed a computer that used VLIW processors to execute instructions up to 1024-bit in size. Along with these computers Multiflow also designed a compiler system that used trace based scheduling to extract ILP from programs. 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 [8].

John Fisher has designed the VEX ISA as an example of VLIW type processors [9]. His work includes the design of a VLIW processor and ISA and a compiler system that generates code for this processor.

Currently two different compilers exist that target the ρ-VEX processor: the VEX compiler 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 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 subjects: Code quality, support, languages support, backend supported and customization possibilities.

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 not possible because compiler is proprietary and closed source. Because of this expanding the functionality of the compiler is impossible.

GCC:

- **Code quality:** Excellent code quality with performance approaching that of commercial compilers (CITATION NEEDED).
- **Support:** There is a very active development community around GCC.
- **Front-end:** GCC supports a large number of programming languages including C, C++, Fortran and Java
- **Back-end:** Supoort exists for a large number of processors including x86, ARM, MIPS and ofcourse VEX
- **Customization:** Because GCC is open source the compiler can be customized to support new passes, optimizations and instructions.

Unfortunately GCC has a number of disadvantages that need mentioning.

- **VEX code quality:** The 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 correclty when compiled by GCC. Some programs are unable to be compiled by GCC.
- **VEX reconfiguration:** The current GCC VEX compiler does not support runtime reconfiguration. The compiler has been set to a 4 issue width ρ-VEX and this cannot be changed without rebuilding GCC.

- **Bloated:** GCC consists of millions of lines of code and is arguable one of the most complex programs in existence. This makes understanding GCC and developing for GCC very hard.

- **Complexity:** GCC is written in C. Design is complex, not very modular and documentation is not very good. Different parts of the compiler are linked in a complex way and it is very difficult to obtain a general picture on how the compiler operates. Because of the complexity it is difficult to achieve high performance in GCC.

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 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++. The GCC front-end was used to translate programs into LLVM compatibale intermediate representation. Around 2005 the Clang project was started which aimed to replace the GCC front-end with an independent front-end with support for 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 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:** Charachteristics of the ρ-VEX processor should be reconfigurable during run-time.

1.3 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 width VLIW
  - Code generation
  - Assembly emitter
- Add support for reconfigurability
  - VEX machine description
  - Reconfigure LLVM during runtime
- Optimize performance
  - Instruction selection
  - Hazard recognizer
  - Register allocator

1.4 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. This chapter will present how a back-end is implemented in LLVM. We will show how code is transformed from the LLVM Intermediate Representation 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. Problems that have been found are demonstrated and we will show how these problems have been solved 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, optimization levels and other metrics. A conclusion and recommendations for future research is presented in chapter 6.

Background 2

In this section we will explore the background of the VEX system and of the LLVM compiler. This section 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 coded 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 [3]. 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 shown in Figure 2.1 [10]. The core uses a five stage pipelined design. There are multiple functional units with different functionality, 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.

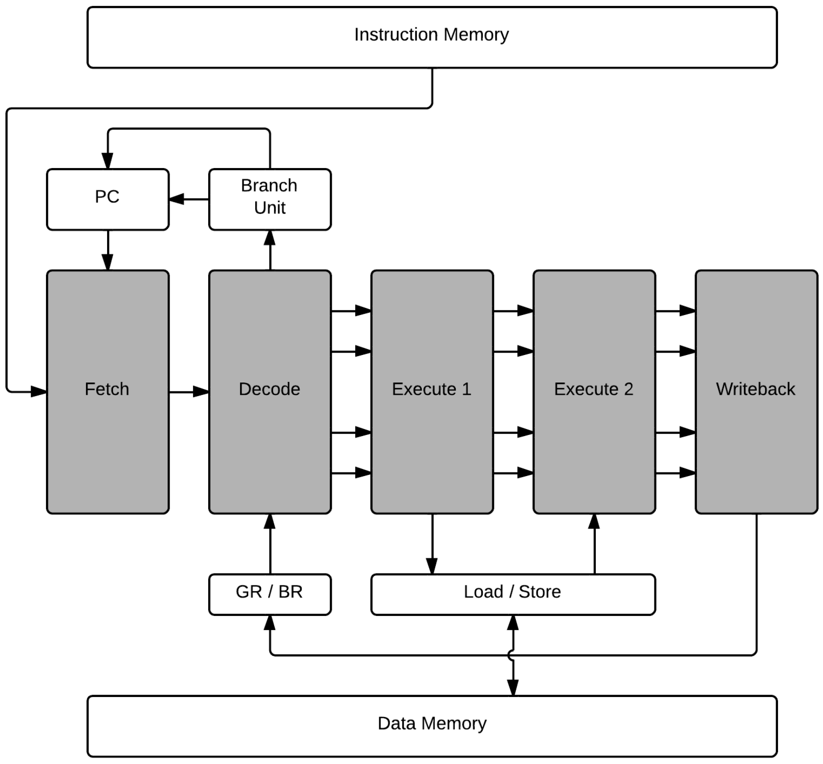

Figure 2.1: ρ-VEX architecture

- **Execute 2 stage:** This stage produces results for MUL operations. The ρ-VEX processor uses $32 * 16$-bit multipliers that have two stages. 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.

2.1.2 ISA

The assembly format for VEX instructions is shown in Figure 2.2 [2]. 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 a operations borrowing scheme. Each operations 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

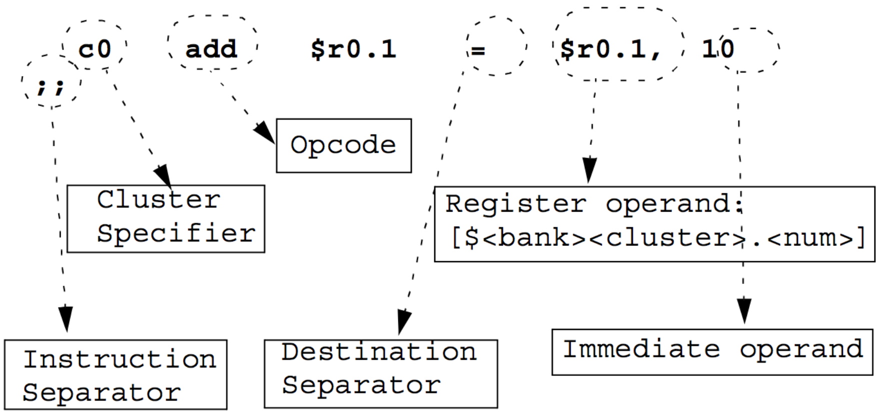

Figure 2.2: ρ-VEX instruction format

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 builtin $16 * 32$-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, long 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. 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-$r0.10`	Scratch	**Arguments and return values:** Arguments that do not fit in the registers are passes using the main memory.
`$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 [2]

- **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 shown in figure 2.3. The compiler uses a number language specific front-ends, an optimizer and target specific backends. This modular design enables compiler designers to work on different parts of the compiler as a separate part. Support for a new processor can be added by building a new back-end. The existing front-end and optimizer can be reused for the new compiler.

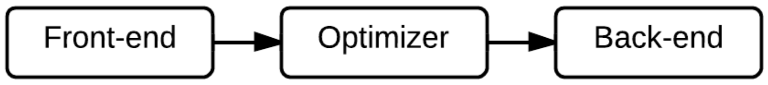

Figure 2.3: Basic compiler structure

The front-end 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 grammer the sequence of tokens is transformed into a parse tree which represents the structure of the program.

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 front-end.

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 (loop vectorization) can be enabled.

The back-end optimizes and generates code for a specific architecture. The LLVM IR is transformed into processor specific assembly instructions, allocates registers and schedules code for better performance.

2.2.1 Front-end

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 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 front-ends and supported languages.

Clang has been developed to allow LLVM to operate independently of GCC. Clang is a front-end supporting C, C++ and ObjC. The front-end 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 [11].

2.2.2 LLVM IR

The front-end transforms a source code into the LLVM internal 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 as human readable text, in memory and in binary form [12].

The LLVM IR is designed to expose high-level information for further optimization. Examples of high-level information include dataflow through 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 bitwidth 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 shows an example program in C. The equivalent LLVM IR representation is shown in figure 2.2.

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

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

Listing 2.1: C example program

```

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.

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 develops more parts of the backend description should be integrated into the backend.

Figure 2.4 shows the basic codegeneration 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.

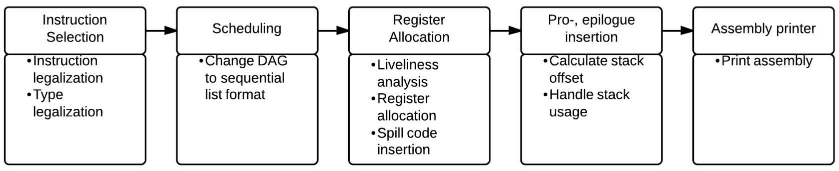

Figure 2.4: Basic codegeneration process

2.2.4 Scheduling

The LLVM compiler uses basic blocks to schedule instructions. A basic block is a block of code which has exactly one entry point and one exit point. This means that no jump instruction exists with a destionation 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 and are used to store which constants are spilled to memory, the objects that are allocated on the stack, target specific function information and which registers are used respectively.

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 width 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 an actual ρ-VEX processor is simulated and is used for execution. The disadvantage of using Modelsim is the performance. Simulation of an executable will take a long time because the complete processor is simulated.

Verification of the compiler will be performed by writing test programs that generate certain instruction sequences. The output of these test programs will be compared to the expected result to check for errors in the backend.

Writing testprograms that have a high coverage of all the possible output patterns is impossible because of this a second round of verification will be performed by compiling benchmark programs that execute common programs. The output of the benchmarking programs will be compared to expected outputs to check for errors in the compiler.

2.4 Conclusion

This chapter discussed the background of the ρ-VEX processor and the basics of the LLVM compiler framework. Instructions that are supported by the ρ-VEX processor have been shown, the run-time architecture has been discussed and the register layout of the ρ-VEX processor has been discussed. The background information on the LLVM compiler should provide for a basic understanding on how the compiler operates and what parts need to be implemented to build a ρ-VEX backend.

Finally we have also discussd how the LLVM compiler will be verified. The verification step is extremely important to determine wheter 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. 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”. Superclasses can be described to a subclass can derive structure from another class. 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. For example processor instructions could 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 such as liveness information, instruction pattern and scheduling information.

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

The rvexInst class can be used for instructions that operate on three operands such as add, sub, and mult. Common features of these instructions are described in the class `ArithLogicR`. This class describes the instruction pattern to match and the instruction string to emit.

```
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>
{
}
```

Finally, this class is used to describe processor instructions such as the add instruction. The instruction is described as using the class `ArithLogicR` with a certain instruction string, instruction pattern and other information needed during compilation.

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

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 `vexReg` class is used to define all ρ-VEX registers.

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

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;
}
```

Each physical register is defined as an instance of one of these classes. For example, R5 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]>;
```

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)
  )>;
```

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 a lot of trouble identifying the correct instruction patterns for compare instructions. The compare instructions operate on two CPURegs instructions and produce a result in BRRegs or in CPURegs. For example consider the following pseude instruction patterns:

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

When an operation works on 32 bit operands and produces a result 1 bit wide LLVM will try to truncate the input operands to the final operand with even though this is not necessary for the ρ-VEX ISA. 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. A processor functional unit executes each instruction. For this example we will assume that the ρ-VEX processor is a 4 issue width machine.

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

Each instruction is associated with an instruction itinerary. An instruction itinerary groups 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. Each instruction itinerary is derived from an instruction stage with certain properties. 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<IIHiLo             , [InstrStage<1,  [P0, P1, P2, P3]>]>,
  InstrItinData<IIImul             , [InstrStage<1,  [P0, P1, P2, P3]>]>,
  InstrItinData<IIAluImm           , [InstrStage<1,  [P0, P2]>]>,
  InstrItinData<IIPseudo           , [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
- Subtarget features

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, that are performed in order.

3.2.1 Instruction selection

The instruction selection phase is completed in the following steps

- **Build innitial DAG:** DAG build from LLVM IR instruction that contains illegal types and instructions.
- **Legalize instructions:** Illegal instructions are expanded and replaced with legal instructions.
- **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 processor instructions.

The instruction selection passes are contained in `rvexISelLowering` class and `rvexISelDAGToDag` class. The `rvexISelLowering` class gives a high level description of the operations that are supported by the target processor. The class can describe how the compiler should handle each LLVM IR instruction through 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.

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.

3.2.1.1 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 and returns a result of type i[2*N]. Through expansion this instruction will be transformed into two mult instructions that calculate the low part and the high part separately.

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

3.2.1.2 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);
```

3.2.1.3 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.

3.2.2 New instructions

The ρ-VEX processor supports some instrruction that have no equivalent LLVM ISD operation. These instructions include `divs`, `addcg`, `min`, `max`, and others. The rvexISelLowering class is used to define when these instructions need to be used.

For example consider the divide operation. The ρ-VEX processor supports division in steps using the divs instruction. The divs instructions executes a single step of a division algorithm. The algorithm that performs division and that uses the divs instruction is implemented in the rvexISelLowering class as a custom expansion of the LLVM div instruction.

At this point the compiler knows how to use the custom instructions but still does not know how to select and match the custom instructions. The tablegen instruction description files are used to define the custom instructions. First the operand types for a custom instruction are defined. The following code shows the definition for the `addcg` instruction that has five operands.

```
def SDT_rvexAddc          : 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 SelectionDAG node 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>;
```

With the custom instruction named we can now define a custom instruction class. Note that the instruction pattern that is used for matching is empty. This is because the current version of tablegen has no support for instructions that define multiple values. Custom selection will be used for matching of these instructions.

```
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"),
     [//EMPTY], itin, type> {
  let shamt = 0;
  let isCommutable = isComm;  // e.g. add rb rc =  add rc rb
  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 `divs` 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;
  }
```

Some 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 SHX-ADD 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)>;
```

3.2.3 Floating-point operations

ρ-VEX processor does not natively support floating-point 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.

3.2.4 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.

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.

3.2.5 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 DAG. Multiple register allocation algorithms are available. All algorithms operate on the liveness information.

3.2.6 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
;;
```

A Post Register Allocation scheduler is used to reschedule code and insert nops when required. The Post-RA scheduler uses a the `ScheduleHazardRecognizer` class to recognize hazards. The `ScheduleHazardRecognizer` class uses the pipeline description to recognize structural hazards such as pipeline or resource constraint. If a hazards has been found the Post-RA scheduler will try to insert an alternative instruction. If this is not possible the processor pipeline will be stalled. The ρ-VEX processor has no support for hardware stalls so `nop` instructions should be inserted when a hazard occurs.

The ρ-VEX processor has a 1 cycle delay between defining a branch register and using a branch register. The hazard recognizer 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 not dependent on the proceeding instructions. Unfortunetely this proved impossible to implement in the current version of the LLVM compile because it is not possible to specify a delay between specific instruction classes

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:

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

3.2.7 Prologue and epilogue insertion

After the register allocation pass prologue en epilogue functions can be calculated. The prologue and epilogue pass is used calculate the correct stack offset for each variable.

Code is inserted that reserves room on the stack and saves / loads variables from the stackframe.

3.2.8 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 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 width VLIW processor but with only units supporting load / store operations. The DFA can model this pipeline and guarantee only 1 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

By disallowing RAW hazards in instruction bundles, binaries can be generated that can execute on any ρ-VEX processor irrespective of issue width. For example consider the following instruction packet. The RAW hazard will occur on the r0.8 register.

```
  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
;;
```

Execution of this processor on a 4 issue width processor will be fine but if this instruction is executed on a 2 issue width processor a problem will arise. The contents of register r0.8 will be changed during execution of the first bundle and execution of the second bundle will produce an invalid result.

This bundle should be split into two instructions during compilation. This can be achieved by checking for RAW hazards inside a bundle during the VLIW packetizer pass. The VLIW packetizer will detect these hazards and split the instructions into different bundles so correct execution is guaranteed.

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 during runtime.

Currently runtime reconfiguration is supported through subtarget support. Backend features can be enabled and disabled through command line parameters. The ARM backend uses this approach to select between different ARMv7 architectures, floating point support and div/mul support. This approach would 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 will be read parsed and 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 delay associated with a functional unit
- **Instruction itinerary:** Maps an instruction itinerary to a function unit

The parameters are also used to generate the DFA that will track the resource unit. 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.

¡MOET IK HET ALGORITHME VERMELDEN EN UITLEGGEN?¿

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 during anypoint in the compilation process. The `Is_Generic_flag` is used during the `rvexVLIWPacketizer` pass and during the liveliness calculation pass.

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

The `InstrItinerary` parameter describes which type of instruction maps to which `Stage`. The first value describes which Stage an instruction is going to use, the second parameter describes when the result of the operation is available.

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

3.4 Conclusion

In this section we have shown how the ρ-VEX backend for the LLVM compiler has been implemented. A short description has been provided on how the Tablegen features are used for description of the a backend. We have shown how the ρ-VEX processor has been described in Tablegen and how the different passes have been implemented. All the phases that are involved with code generation have been discussed.

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.

We have also shown how the performance of the backend has been improved by implementing custom scheduling features and a custom register allocator.

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 Generic binary optimization

Research has shown [13] 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 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.

4.1.1 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 on 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.

A second approach is to form temporary instruction bundles during the scheduling pass and for the scheduler to provide register allocation hints to the register allocator. This is a similar approach to the machine scheduler pass that from the Hexagon target. Unfortunately this proved impractical because liveliness information was not available during the scheduling pass. The Hexagon machine scheduler uses the temporary bundles and certain register allocation metrics to optimize the instruction ordering but the extra information is not passed to the register allocator.

For this paper the choice has been made to make changes to 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
;;
```

4.1.2 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 `ExtendToUses` method of the `LiveRangeCalc` class is used to update the SlotIndex to match the latest use of a virtual register. The SlotIndex class also provides method that give information on the MachineBasicBlock (MBB) in which the virtual register is used.

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.

4.2 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 instruction. Values larger then 8 bit values borrow space from the adjecant ρ-VEX instruction. The following code examples show the maximum amount of instruction in a packet for a 4 issue width 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
;;
```

```
  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 arithmatic instruction can use large immediate values but also load and store instructions to represent the address offset.

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

4.2.1 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.

Unfortunetely 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 and to recognize large immediate values during the packetization pass.

4.2.2 implementation

During the packetization pass each instruction that uses an immediate value is checked before it is bundled in an instruction bundle. The packetizer determines the size of an immediate value and reserves an extra resource in the DFA when a large immediate value is used.

4.3 Conclusion

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

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

In this chapter we are going to discuss the verfication of the LLVM compiler and the performance of the binaries that are generated with the LLVM compiler.

5.1 Verification

The correct operation of a compiler is extremely important

5.2 Benchmark results

Comparison to GCC and HP-VEX
    Different issue widths
    Different optimization levels

5.3 Generic binary

Comparison to GCC
    Different issue widths

5.4 Conclusion

asd

Conclusion 6

Bibliography

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

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

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

[4] 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.

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

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

[7] 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

[8] 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.

[9] C. Y. Joseph A. Fisher, Paolo Faraboschi, *Embedded computing: A VLIW Approach to Architecture, Compilers and Tools.* Elsevier, 2005.

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

[11] Clang, "Clang - features and goals."

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

[13] S. W. Anthony Brandon, "Support for dynamic issue width in vliw processors using generic binaries," *EDAA*, 2013.

List of definitions

.. ...

LLVM Quickstart guide A

asd

A.1 LLVM toolchain

asd

A.2 XSTsim

asd

B LLVM Development guide

asd

B.1 Building LLVM from source

asd