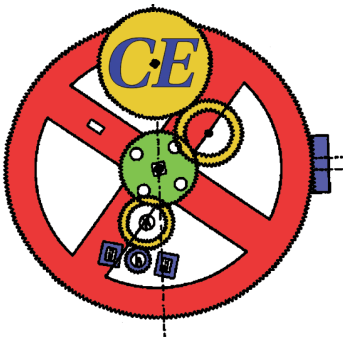


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2014

MSc THESIS

LLVM based ρ-VEX compiler

Maurice Daverveldt

Abstract

CE-MS-2014

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 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	:	
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

Acknowledgements

Maurice Daverveldt
Delft, The Netherlands
January 27, 2014

1 Introduction

1.1 Motivation

In 2008 Thijs van As designed the first version of the ρ-VEX processor [1]. 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 [2] 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.

VLIW differs from multiple issue 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 OoO does not need to be implemented. The compiler is responsible for finding all the parallelism and for scheduling code in an efficient way. This also enables the compiler to execute additional optimization because the compiler has got a higher level view of the code that needs 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 [3]. 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 [4].

1.1.1 Futore of ρ-VEX

asd

- Runtime reconfiguration of processor and of compiler.
- Possible JITing of code
- Generic binaries, one size fits all

1.2 Problem statement

Currently both HP-VEX and GCC can be used to generate code for the rvex 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 Apple started the Clang project (CITATION NEEDED) 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".

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.

In addition research will be done on the following subjects:

- **asd:** asd

1.3 Methodology

The following steps need to be completed for successful implementation of a rvex LLVM compiler.

- Research rvex 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. The background of the ρ-VEX processor and of the LLVM compiler will be explored in the chapter 2. In chapter 3 will discuss how the ρ-VEX compiler was implemented. This chapter will present how a back-end is implemented in LLVM. Additional optimization and added functionality is also discussed in this chapter. Chapter 4 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 5.

2 Background

2.1 VEX System

asd

2.1.1 Architecture

asd

2.1.2 ISA

asd

2.1.3 Registers

asd

2.1.4 Run-time architecture

asd

2.2 LLVM Compiler infrastructure

LLVM is based on the classic three-stage compiler architecture shown in figure 2.1. 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.

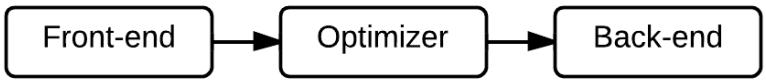


Figure 2.1: Basic compiler structure

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 LLVM

asd

2.2.1.1 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 [5].

2.2.1.2 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 [6].

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.

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

2.2.1.3 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 for a backend to operate. As LLVM develops more parts of the backend description should be integrated into the backend.

The following sequence is completed to translate LLVM IR into target specific assembly language:

1. Instruction selection
2. Scheduling
3. Register allocation
4. Prologue / epilogue insertion
5. Assembly printing

3 Implementation

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 rvexReg 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 rvex 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 rvex 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 rvex 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 rvex 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

asd

3.2.1 Instruction selection

asd

3.2.2 Scheduling

asd

3.2.3 Register allocation

asd

3.2.4 Pipeline description

asd

3.2.5 VLIW Packetizer

asd

3.2.6 Assembly emitter

asd

3.3 LLVM Improvements

asd

3.3.1 Reconfigurable compiler

asd

3.3.1.1 Machine description

asd

3.3.1.2 Compiler run-time reconfiguration

asd

3.3.1.3 Optimize VLIW scheduling

asd

3.3.1.4 Generic binary support

asd

4 Results

4.1 Verification

asd

4.2 Benchmark results

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

4.3 Generic binary

Comparison to GCC
- Different issue widths

5 Conclusion

Bibliography

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

[2] 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.

[3] 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

[4] 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.

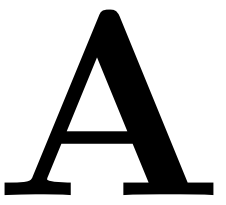
[5] Clang, "Clang - features and goals."

[6] V. A. Olatunji Ruwase, Chris Lattner and G. P. David Koes, "The llvm compiler framework and infrastructure (part 1)," Presentation.

List of definitions

.. ...

LLVM Quickstart guide



asd

A.1 LLVM toolchain

asd

A.2 XSTsim

asd

LLVM Development guide

B

asd

B.1 Building LLVM from source

asd