

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

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
Codenummer : 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	vii
List of Tables	ix
List of Acronyms	xi
Acknowledgements	xiii

1 Introduction	1
1.1 Motivation	1
1.1.1 Origin and history	1
1.1.2 Future of ρ -VEX	1
1.2 Problem statement	1
1.2.1 Goals	1
1.2.2 Subgoals	1
1.3 Methodology	2
1.4 Thesis overview	2
2 Background	3
2.1 VEX System	3
2.1.1 Architecture	3
2.1.2 ISA	3
2.1.3 Registers	3
2.1.4 Run-time architecture	3
2.2 LLVM Compiler infrastructure	3
2.2.1 LLVM	4
3 Implementation	7
3.1 Tablegen	7
3.2 Code generation	7
3.2.1 Instruction selection	7
3.2.2 Scheduling	7
3.2.3 Register allocation	7
3.2.4 Pipeline description	7
3.2.5 VLIW Packetizer	7
3.2.6 Assembly emitter	7
3.3 LLVM Improvements	7
3.3.1 Reconfigurable compiler	7

4	Results	9
4.1	Benchmark results	9
4.2	Generic binary	9
5	Conclusion	11
	Bibliography	13
	List of Definitions	15
A	LLVM Quickstart guide	17
A.1	LLVM toolchain	17
A.2	XSTsim	17
B	LLVM Development guide	19
B.1	Building LLVM from source	19

List of Figures

2.1	Basic compiler structure	3
-----	------------------------------------	---

List of Tables

List of Acronyms

Acknowledgements

Maurice Daverveldt
Delft, The Netherlands
January 20, 2014

Introduction

1.1 Motivation

In 2008 Thijs van As designed the first version of the rvex 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 rvex 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.

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

1.2 Problem statement

asd

1.2.1 Goals

asd

1.2.2 Subgoals

asd

1.3 Methodology

asd

1.4 Thesis overview

asd

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.

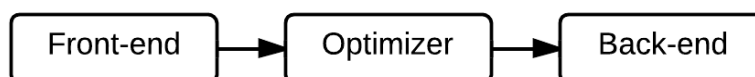


Figure 2.1: Basic compiler structure

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 grammar 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

Implementation

3.1 Tablegen

asd

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

Results

4.1 Benchmark results

Comparison to GCC and HP-VEX

Different issue widths

Different optimization levels

4.2 Generic binary

Comparison to GCC

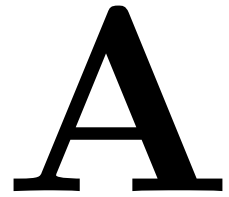
Different issue widths

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

.. ...



asd

A.1 LLVM toolchain

asd

A.2 XSTsim

asd

asd

B.1 Building LLVM from source

asd