FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	TinyC Optimizing Compiler
Student:	Bc. Martin Slávik
Supervisor:	Ing. Petr Máj
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

Design and implement an intermediate representation and an optimizing compiler for the tinyC language and tiny86 target VM using the methods from the NI-GEN course. The compiler must correctly translate the entirety of the tinyC language and generate a correct and non-trivial t86 assembly. Implement intra-procedural optimizations and inlining in the optimizer and instruction selection and register allocation using selected techniques from NI-GEN course in the backend part. The code must be written in idiomatic C++ consistent with the rest of the tinyverse platform, be clearly structured and well documented so that it can be used as a reference implementation in the NI-GEN course.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 10 December 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

TinyC Optimizing Compiler

Bc. Martin Slávik

Department of Theoretical Computer Science
Supervisor: Ing. Petr Máj

May 3, 2023

Acknowledgements

I would like to thank my supervisor Ing. Petr Máj for his patience and valuable advice during the creation of this thesis.

Declaration

In Prague on May 3, 2023 . . . .

Citation of this thesis

Slávik, Martin. *TinyC Optimizing Compiler.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Práce pojednává o tvorbě kompilátoru, a jeho vnitřních částí pro výukové účely.

Práce má za úkol vytvořit kompilátor, který přeloží programy zapsané v jazyce TinyC do instrukční sady definované virtuální architekturou tiny86. TinyC je podobný jazyku C s menší sadou typů a redukovanou funkcionalitou. Tiny86 virtuální stroj je inspirovaný x86 architekturou s jednodušší instrukční sadou, za účelem lepší srozumitelnosti. Instrukční sada tiny86 zachovává zajímavé a důležité vlastnosti, které architektura x86 nabízí. Během překladu kompilátor využívá optimalizační techniky pro dosažení lepšího výkonu kompilovaného kódu. Práce porovnává výsledky naivní kompilace a kompilace se zapnutými optimalizacemi.

Klíčová slova kompilátor, optimalizační kompilátor, TinyC, překladač, alokace registrů, výběr instrukcí

Abstract

This thesis is about compiler creation and its internal parts for educational purposes.

It should compile programs described with the TinyC language to the instruction set specified by the virtual architecture tiny86. TinyC is a C-like language with a smaller subset of types and a limited set of features. The tiny86 Virtual Machine is inspired by x86 architecture but has a simpler instruction set, so it is more understandable for students. The tiny86 instruction set preserves interesting and important features that the x86 architecture offers. Throughout the compilation, optimization techniques are used to achieve better performance of the compiled code. The thesis compares the naïve compilation and compilation with optimization techniques turned on.

Keywords compiler, optimizing compiler, TinyC, translator, register allocation, instruction selection

Contents

Listings

List of Figures

List of Tables

Introduction

In the beginning, computers were programmed by directly specifying the operations that the computer was capable of. These operations were encoded as numbers in a representation called the machine code. The construction of such programs was straightforward. The programmer had precise control over the target system's behavior, and the target machine's full potential could be accessed. On the other hand, precise control means that the programmer has to control everything and specify it in the machine code. Unfortunately, such a description of an algorithm, while perfect for the machine, is almost useless for humans because the target operations are far too primitive to describe the complex idea being developed. The first step towards expressing more readable programs for humans was to replace the numbers that encode the operations with short mnemonics, which led to the Assembly language, which was better because the instructions were easier to decipher and the programmer could see what the instruction did. But still, the idea of the program was hidden behind the Assembly instructions.

The next evolution came with an abstraction of the operations used in programming, such as loading and storing a variable. This abstraction led to high-level languages. High-level languages use constructs that help programmers avoid repeating code and separate themselves from low-level details that machine code carries with it. The Example 0.1 on the left shows a program written in a high-level language, whereas, on the right, there is an Assembly representation of the same program. There are many instructions, and some parts are repeated. That shows why a high-level representation is more convenient for programmers because they do not need to repeat themselves.

To grasp the program's structure, the programmer can use constructs like functions for specifying reusable code that can vary in inputs, but the core idea stays the same. For these constructs to work, they need to be translated into a correct underlying machine code. Conversion back to low-level details can be done automatically by a program. This program is called a compiler.

```
; Assembly code                     // C code code
main:       # @main
  push  rbp                         int main(){
  mov   rbp, rsp
  sub   rsp, 16
  mov   dword ptr [rbp - 4], 2023   int year = 2023;
  mov   dword ptr [rbp - 8], 0      int userYear = 0;
  lea   rdi, [rip + .L.str]         printf("What year were
  mov   al, 0                                 you born?");
  call  printf@PLT
  lea   rdi, [rip + .L.str.1]       scanf("%d", &userYear);
  lea   rsi, [rbp - 8]
  mov   al, 0
  call  __isoc99_scanf@PLT
  mov   eax, dword ptr [rbp - 4]    int userAge = year - userYear;
  sub   eax, dword ptr [rbp - 8]
  mov   dword ptr [rbp - 12], eax
  mov   esi, dword ptr [rbp - 12]   printf("You are %d years old."
  lea   rdi, [rip + .L.str.2]            , userAge);
  mov   al, 0
  call  printf@PLT
  xor   eax, eax                    }
  add   rsp, 16
  pop   rbp
  ret
.L.str:
  .asciz  "What year...born?"
.L.str.1:
  .asciz  "%d"
.L.str.2:
  .asciz  You are %d years old."
```

Listing 0.1: The difference between high-level (on the left) and low-level(on the right) languages of the same program.

Initially, a brand new compiler had to be created for every language. A simple program that found a statement and converted it to specific target instructions. However, as these compilers evolved and many features were added because programmers realized that compilers could do much more than replace the constructs back. Decisions about the target's resource usage must be made during the compilation. For example, the target machine can have *registers*. Registers serve as storage where the program values are held, and the target machine uses them for addressing. Instructions can copy, load from, and store to the registers.; for example, an `ADD` instruction loads from two registers, and the result of the addition is stored in a register. Values can also be stored in memory, but the operations working with memory are expensive, therefore, it is better to use them as little as possible.

It is difficult and labor-intensive for programmers to specify which registers should hold which values (and when during the program) while, in the target, there is a limited amount of these registers. Therefore, a machine will solve these problems. Unfortunately, it is very hard or sometimes impossible for all programs. Designing these algorithms to be fast and effective and resolving similar problems arising from optimizing parts inside the compiler is the hard part of creating a compiler. Furthermore, optimization is the next part of why the program compilation is considered challenging because the compiler should not only give a working program but a faster equivalent that does not waste the target resources. The compiler which gives better (faster and more efficient) programs is called the optimizing compiler.

Although constructing a compiler is hard, it is beneficial for programmers because the compiler reduces the amount of work that they need to put into creating a program. Moreover, it prevents programmers from making mistakes in parts that the compiler manages nowadays since it is the responsibility of the compiler, and usually the compiler gives better results than deciding this manually for larger programs. It is also very beneficial because the programmer can focus more on the algorithmic part of the programming; therefore, the development of programs can be faster than before.

TinyVerse

The task of this Master's thesis is to build an optimizing compiler, which can be used in a course NI-GEN, where students are required to build a naïve compiler. This thesis presents a compiler with more features so the students can see what is the difference between using various optimization techniques and none — compiling entirely naïvely.

The TinyVerse project helps with compiler teaching. It is a whole package that can compile and run compiled code. All languages and targets share the compiler functionality. The languages and targets then provide the language- and target-specific functionality, such as front-ends, Optimizers – middle-ends, and back-ends. The whole system is designed so that its parts can be swapped at compile time to suit different languages, targets, optimizers, etc.

Two parts of TinyVerse are essential for this thesis.

The first is the TinyC language, which we will compile. TinyC is a C-like language with a smaller subset of types (int, char, double, arrays and structs, function pointers, and pointers) and a limited set of features. TinyC's semantics and grammar are simpler, which makes TinyC not a strict subset of C. The language specification can be found in chapter B located in the appendix of this thesis.

The second is the package t86 which contains tiny86 virtual machine — the target to which the compiler from this thesis compiles. Tiny86 was inspired by the x86 Architecture, but with a reduced instruction set so it is easier for students to understand the compilation process. The simplification kept both interesting and important parts of the x86 architecture which are valuable for teaching.

Thesis Overview

This thesis comprises four chapters: Overview, Design and Implementation, Evaluation, and Conclusion.

- The chapter Overview describes how modern compilers are designed. Modern compilers define an internal representation (IR) that serves to represent the source program in a form that is more suitable for the compiler; section 1.2 describes the internal representations — their design, advantages, and disadvantages of various designs. It also describes the composition of regular modern compiler suites, and the most popular compiler suites nowadays.

- The second chapter, Design and Implementation, describes and justifies the ideas behind the algorithms picked for the implementation including the design of the internal representation, description, and examples of the translation performed in the front-end and the ideas behind the algorithms used in the Optimizer and the compiler's back-end.

- The third chapter, Evaluation, contains the evaluation of the optimization changes in terms of effectiveness and the number of instructions that need to be executed.

Chapter 1

Overview of a Modern Compiler

Compilers used to be straightforward programs that only replaced parts of the input code for sequences depending on which target machine the compilation was performed. Making keywords instead of low-level sequences has two significant benefits, the independence from the target machine — the programming no longer requires specific coding instruction and the readability for a programmer. With this evolution came slight problems in the form of question: Values are stored in registers, and who defines which value comes to which register? First languages used hint keywords by which the programmer could specify in which register the value should be, and the compiler's task was to respect the constraints that the programmer gave for a value where it should be. However, as compilers evolved, less and less emphasis was put on that functionality. The task of which value should reside in which register soon became the task of compilers. Like this, more tasks make the outputted programs faster or memory efficient. Generally, compilers are valuable tools for programmers and became more complicated.

Throughout the evolution of compilers, rules were established that define what every compiler should do:

1. The compiler has to generate an appropriate valid output for every valid input. So, the outputted program has to keep its behavior described in the input program.

2. It should warn and terminate the compilation if the input is not valid.

3. Also, the output program should be as effective as possible. Concerning the size or speed of execution of such a program. (This is why compilation is considered challenging.)

The core of every modern compiler of some programming language is often very similar. This similarity comes from the fact that it has the same task of translating and optimizing. It reads the input, performs some optimization, and translates it into or machine code. As the compiler has several tasks to help the programmer with, it is helpful to divide modern optimizing compilers into parts. This separation also has advantages considering reusability when another source language or target machine needs to be supported.

When the compiler parts can be swapped, an interface must be designed. This interface in the program representation is called intermediate representation. The modern compiler parts are the front-end, middle-end, and back-end as shown in Figure 1.1.

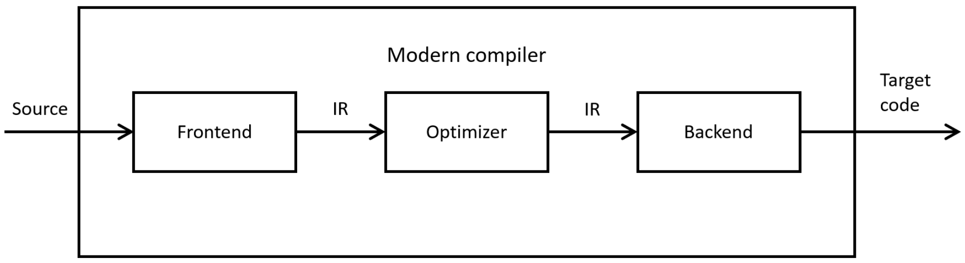


Figure 1.1: High-level overview of a Modern compiler

- *Front-end:* The front-end task is translating the source language to the intermediate representation specific to the compiler.
- *Middle-end:* The middle-end is the core of the compiler. It performs all operations specific to the Optimizations, making the outputted program more efficient.
- *Back-end:* The back-end's task is a translation to the specific target; when a different target has to be used, the target has to be changed. It contains all parts of the compiler dedicated to the specific target.

When compilers need to translate source code described in a different source language or compile to a different target, they differ only at the beginning of the process; thus, a front-end has to be changed to support the language, or the second difference is at the back-end, where a different target and its instruction set is available that means a change in code generator and target optimizations.

The next part describes the most significant parts of a compilation with more details.[1]

Front-end with the task to understand and transform the source program, find errors in it, and translate it into IR can be separated into parts:

- *Lexical Analysis* The task of lexical analysis is to read and separate its input into tokens defined by the input language. For example,

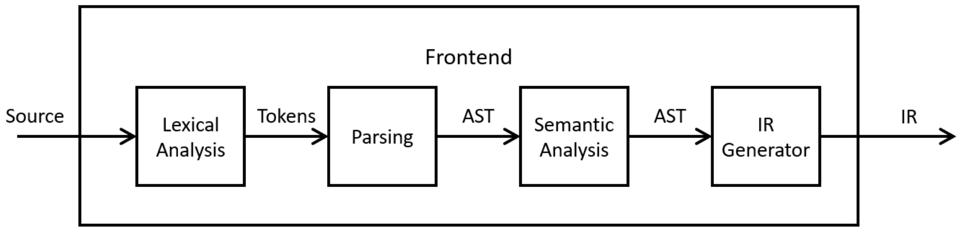


Figure 1.2: Design of front-end internals of a modern compiler

`if`, `{`, `5`. This part aims to have less work while implementing syntax analysis.

- *Syntax Analysis* The task is constructing an abstract syntax tree (AST) structure from the tokens produced before. This step also checks the validity of the input code and that the AST nodes can be created from the input.
- *Semantic Analysis* checks the correctness of the AST. The main part is the Typechecker, which checks if the program is type sound – if the used operations support the types supplied to them. Semantic analysis can check the validity of working with variables if it is declared before use. These first three steps also generate errors for the error handler that informs the programmer where a mistake happened or warnings about some part of the code will work but maybe not as intended.
- *IR generation* After the correct AST is constructed and passed to the IR generation, the IR is generated. The form of IR is specified by the compiler and is useful for various analyses. The form is usually of an SSA or a three-address code type.

Target independent Optimizations perform the majority of optimization generally in separated passes. This part belongs to *middle-end.* The advantage is that these optimizations are neither source- nor target-dependent.

The *back-end*'s task is to translate the IR into the target code. It consists of two parts:

- *Code Generator* The next step is to translate the independent IR into a code mainly designated for the target in the form of a machine code. For this translation, instructions and registers need to be selected. Both processes play a crucial role in the efficiency of the target code.

- *Target-dependent Optimizations* The last part eliminates the sub-optimality generated in the previous step and the target-dependent optimization.[1]

If there is a need for a compilation of a large program, it is possible to multiply all three parts of the compiler so that they can run in parallel, but then an additional step called linking is required to build the final program. For this purpose, objective files are constructed by the back-end instead of a full-fledged executable. The linker takes object files and patches dependencies between them.

1.1 Front-end

The front-end translates the input programming language to the Internal Representation. Translation can be done naïvely in case each keyword or construct in language has its specific form in Internal Representation. If the IR instruction is not present, a transformation into a combination of IR instructions that achieves the same result is necessary.

1.1.1 Lexical Analysis

Lexical analysis is the first phase of a compiler. The lexical analyzer breaks the input into a series of tokens by removing any whitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer.

Tokens are of several types:

- *Keywords* are words reserved by the source language with special meanings such as `int`, `break`, or `while`.
- *Identifiers* are words that meet requirements defined by the language, usually used for naming constructs supported by the source language.
- *Operators* The special non-alphabetical characters define *operators.*
- *Numbers* are integral or decimal, created by digits.
- *Strings* and *Characters* are sequences of characters usually surrounded by language-defined characters.

[1]In some literature, the last two steps are swapped or merged together. But the work that needs to be done persists.

1.1.2 Syntax Analysis

Syntax analysis, also known as parsing, is the phase where the compiler analyzes the structure of the source code and checks for syntax errors. The primary goal of syntax analysis is to ensure that the code conforms to the rules of the source language's grammar and does not include any (grammar) errors.

During the syntax analysis phase, the parser analyzes the source code broken down into tokens obtained from the lexical analysis. The parser checks that the sequence of tokens is valid according to the source language grammar. The parser generates an abstract syntax tree (AST), which represents the code structure without the details of the tokens. Syntax analysis can detect syntax errors, such as missing semicolons, mismatched parentheses, or invalid expressions. When syntax errors are detected, the parser generates error messages that help the programmer correct the errors.

1.1.3 Semantic Analysis

Semantic analysis is the phase where the compiler analyzes the meaning of the source code and checks for semantic errors. Errors in this part that can appear are grammatically correct but do not make sense. For example, the program is not type coherent. The primary goal of semantic analysis is to ensure that the code is semantically correct and can be executed without any issues.

1.2 Intermediate Representation

Intermediate representation, mostly called IR, represents the input program and other compiler-generated information that the compiler needs to work with and understand. IR is a representation independent of the source code and target machine. It resolves tasks such as how to represent the control flow, machine operations, types, etc.

The advantages of using this representation should be:

- Easy to process, such as create, read, write, traverse, or transfer.

- Easy to optimize, for example, allowing target-independent low-level optimizations.

1.2.1 Types of IR

Types of IR can be conceptually divided by the form of the representation — main classes are graphical and linearized. *Abstract syntax trees* or *control flow graph* belong to the graphical class. *Stack based* and *Three-address code* belong to the linear-based class. The linear based are closer to a machine code, therefore, are considered a lower-level. The last is *Hybrid class*, which combines the properties of both mentioned.

Tree Based

Tree-based IR is used to represent the structure and semantics of a program in a hierarchical tree data structure. In this representation, each tree node represents a statement, expression, or other constructs in the source program. It is often pictured with edges representing the relationship between the nodes. In tree-based IR, can be found AST or parse trees.

The advantage is that these types of IR are closer to the source language and can usually express all the source semantics. Source language type systems and variables can be reused. The tree IR is a trivially constructed prom parser. And they are easy to translate, for example, by syntax-driven code generation. Tree IR is great for local optimizations like constant propagation or common sub-expression elimination.

The disadvantage comes from tree representations being far from the target language. Memory efficiency is unsatisfactory because of its poor locality in the naïve form. This representation is unsuitable for global optimizations when they do not provide information about the program's control flow.

Linear Based

The second structurally different type of IR is the linearized one. This representation's structure is a sequence closer to the machine code. Therefore this IR can be considered a lower-level than tree based.

Stack Based

The stack-based IR uses stack structure to address values; therefore, most instructions do not need explicit specification of arguments. They get them from the stack. They represent operations such as pushing to or popping from the stack. When performing an arithmetic operation, binary, for example, the input operands are taken from the top of the stack. The result of an operation is pushed back to the stack.

The most significant benefit of stack IR is simplicity and straightforwardness. The small instruction set belongs to the advantages of this IR because stack IR always works with the top of the stack; there is no need for complex addressing features that require more memory to encode the instruction.

On the other hand, other simple operations, such as code movement, are difficult to perform. Just moving the instructions is not enough; preparing and reorganizing the stack for movement is also necessary.

Three-address Code

In three-address code (3AC), each statement is represented as a sequence of up to three operands and an operator. The operands can be either constants, variables, or temporary values. The difference between temporary value and variable is that temporary value originally does not have a place in memory where it can be stored. So when there is a requirement to store it, extra work is required to prepare the memory. In Example 1.1, you can see how expression `d = (a + b) * 15 + (a + b)` can be translated into the 3AC.

```
_1 =  a +  b
_2 = _1 * 15
 d = _2 + _1
```

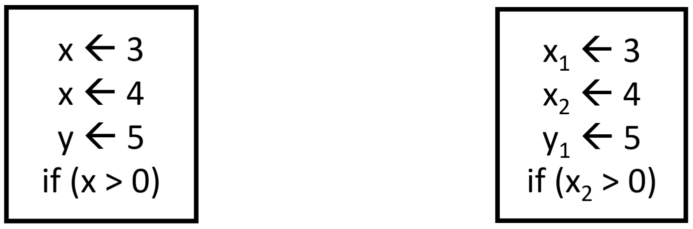
Listing 1.1: Example of three-address code where are used constants, variables, and temporary values

The advantage of the three-address code is its similarity to target instructions because many ISA instructions do not have more than two operands and one output. The next advantage is how easy it is to reorder instructions in a program represented in a three-address code. Instructions can be stored in a list where insertion is easy, and when a few rules are followed when reordering, there is no problem like in stack-based, where instructions need to be added to emulate the stack behavior. Rules such as do not swap instructions after the position of its first use.

1.2.2 Single Static Assignment

In compiler design, static single assignment form (often abbreviated as SSA) is a property of an IR, which requires that each variable is assigned exactly once, and every variable is defined before it is used. Existing variables in the original IR are split into versions, new variables typically indicated by the original name with a subscript so that every definition gets its own version.

When converting to SSA, the tool to preserve the rule is the following: Variable names are appended with indexes that are incremented at each new assignment. Figure 1.3 demonstrates the conversion.



$x \leftarrow 3$
$x \leftarrow 4$
$y \leftarrow 5$
if $(x > 0)$

$x_1 \leftarrow 3$
$x_2 \leftarrow 4$
$y_1 \leftarrow 5$
if $(x_2 > 0)$

Figure 1.3: Example of conversion to SSA Form with the use of indexes.

However, when using this technique, a problem occurs when translating expressions with arguments dependent on the program's control flow. It is shown in Figure 1.4. The solution is to introduce a 'Phi' Instruction[2]. The

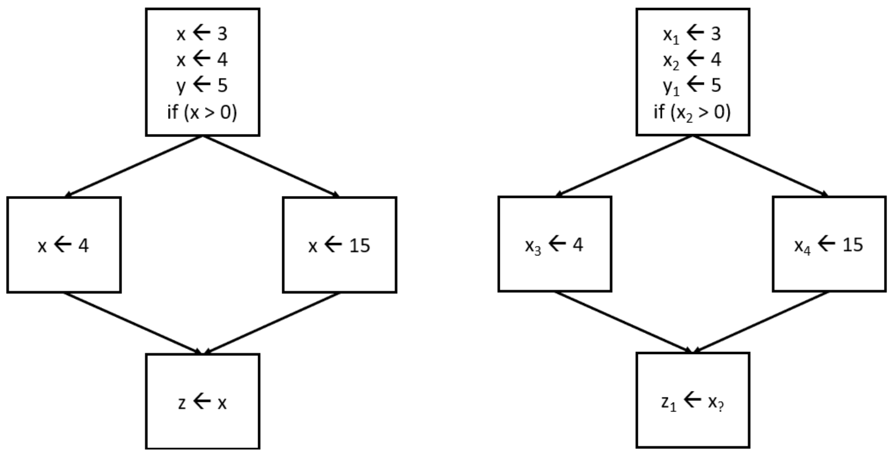


Figure 1.4: Problem occurring in conversion to SSA Form.

'Phi' instruction helps us to use a result from two or more different source instructions in case they are in different Basic Blocks. It manages that each source instruction will eventually share the same register as Figure 1.5 shows.

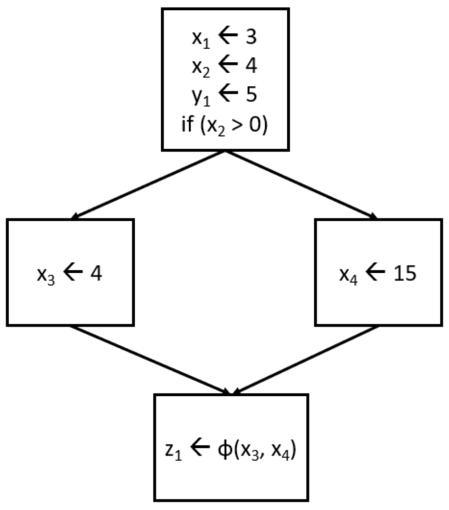


Figure 1.5: Usage of Phi instruction to resolve problems with the conversion.

The reason to use the SSA form is that it brings benefits for analysis. Look at the situation in Figure 1.3; humans can see that the first line is

[2]can be found under *Phi Node* term.

useless because it is overwritten right after the assignment, but a reaching definition analysis must be done for a computer if the code is not in SSA. Other compiler optimization algorithms, such as dead code elimination, common sub-expression elimination, constant propagation, register allocation, etc. are faster and easier to perform when SSA is used.

1.3 Optimizer

The Optimizer has two parts, analysis and optimization. Its task is to analyze the program and find parts that can be rewritten so the program runs faster or to find dead code, which can be omitted, and therefore the program size will be smaller.

This part of a compiler is in the middle-end. Some optimizations require only a pattern to be found, and others require analysis results. These optimizations and analyses run in a loop because the code optimization usually unlocks another optimization. For example, code with tricky dependencies prevents dead code elimination (DCE) from optimizing. However, constant propagation (CP) optimizes a chunk of the tricky part, making it decidable for DCE afterward. The idea is demonstrated in Example 1.2.

```
//1) original code      //2) after CP        //3) after DCE
a = 5;                  a = 5;               a = 5;
b = 0;                  b = 0;               b = 0;
if (b > 4){             if(0 > 4){
  a = 7;                  a = 7;
  b++;                    b++;
}                       }
b +=10;                 b +=10;               b +=10;
```

Listing 1.2: Example showing how entangled can optimization passes be. In code number 1) the ‘then’ branch cannot be easily removed due to the variable b. But after CP showing and replacing b with its constant, dead code elimination can see the guard is false and therefore the ‘then’ branch never executes, thus the code is dead and can be removed.

Pass Scheduler decides how long and what attempts for optimizations are made. The Pass Scheduler is a maintainer of all passes and decides when to start which pass (optimization) will be started, after which one, how many times, and when to end with the optimization. All these decisions are based on the outputs from the analysis that is run between passes or, in better implementations, results of the analysis are updated by the passes because running whole types of analysis could be potentially costly and sometimes even not possible when various types of profile information are stored.

1.3.1 Analyses

The analyses are often divided into two sections, static and dynamic. Dynamic analysis techniques can be used to evaluate the performance and behavior of compiled code, but they are not typically used directly within the compiler itself. The dynamic analysis emulates the execution and collects results. Static analysis involves examining the source code or intermediate representation of a program without the need for emulation or execution.
Every (perfect) analysis goal is to satisfy three properties:

Soudness: Analyses should not miss any errors,

Completeness: Analyses should not raise false alarms,

Termination: Analyses should terminate with a given answer.

Unfortunately, thanks to Rice's theorem, this cannot happen. That is why analyses only approximate and cannot give an exact answer (we wish to have) to every question about the program.[2] The approximation is made so that the answer might be decidable and the results are safe, meaning they must be conservative. For example, we are trying to eliminate an expression from the program. The analysis is asked if the code can be eliminated — its result is not used. The safe answer is that the result is used; we can only eliminate the expression if we are sure it is not used. Otherwise, the optimization would cause errors in the program. Specifying tight criteria can give useless results since the algorithm cannot be sure; on the other hand, too loose criteria can give correct but useless for the Optimizer. So the challenging part about static analysis is to get results that are not useless and do it in a reasonable time and space.

Analyses can also be divided by the range, where the analysis is performed.

An *intra-procedural* analysis is a mechanism for performing optimization for each function, using only the information available for that particular function. Its results are useful for local optimization.

An *inter-procedural* analysis performs analysis on the whole program. Its results are used in global optimizations, which need to consider greater parts of the program. For example, function inlining needs to have information about the whole program. Inlining recursive functions is not beneficial.

The next separation of analysis algorithms is in the direction they perform the analysis, the so-called *forward* and *backward* analysis. This considers how the information is enhanced throughout the analysis.

For example, *liveness analysis*, which gives information about the live ranges of the variables, belongs to the backward analyses. On the other hand, the reaching definition analysis is of the forward type. *Reaching definitions* analysis is useful for dead code elimination. It generally thinks that every variable is not alive at the end. When the first use of the variable is seen (in the backward run – so the last use in the program), the variable is added to the alive set. Then when the definition is seen, it is again removed from the alive set. The result of the liveness analysis is the contents of the alive set at each time.

1.3.2 Optimizations

The optimizations in a compiler are classified into two types: machine dependent and machine independent optimization. The machine-dependent belongs to the back-end part of the compiler and is driven by constraints like what instructions the machine supports or a set of hazards that can occur in the machine. Independent optimizations are located in the middle-end.

The part that makes machine-independent optimization, called Optimizer, has to take results from the analysis described before and performs code rewriting to make the execution of the code faster or to make it more compact in terms of code size.

Generally, optimizations can be divided into global and local ones. Local optimization involves finding the optimal solution for a specific search space region. The global optimization for problems with greater and more complex search space. Therefore, local optimizations are easier to perform because the analysis can give a more specific result due to the reduced context on which the desired change depends. On the other hand, local optimization cannot solve all optimization problems, such as function inlining, there needs to be performed a change across the whole program.

1.3.2.1 Local optimizations

Local optimization is a technique of making a code inside a region more effective for some given criteria. It takes results from the intra-procedural analysis. Generally, the region is a Basic Block since there are no troubles with control flow. To local optimization belongs:

Constant folding involves evaluating constant expressions at compile time instead of at runtime. This optimization can improve the performance of a program by reducing the number of instructions executed.

Algebraic simplifications simplify algebraic expressions in a way that does not change the value of a result but is the most efficient or compact to represent.

Dead code elimination (DCE) is an optimization that identifies and removes sections of code that are no longer needed nor executed during the program's runtime. It improves performance while 'Jump' instructions are removed, reducing the code size overall.

1.3.2.2 Global optimizations

Global optimization can consider the whole functions, where more constraints affect the result, and thus the optimization is more complicated to accomplish.

For example, it makes sense to make constant propagation in a Basic Block and extend it to the whole function while considering the local variables in the whole program. The same applies to function inlining; without the ability to change multiple functions in the program, achieving the desired result would not be possible. To representatives belongs:

Function inlining involves replacing a function call with the body of the called function. This optimization can improve the performance of a program by reducing the overhead of function calls.

Common-subexpression elimination (CSE) involves identifying repeated computations in a program and replacing them with a single computation. This optimization can improve the performance of a program by reducing the number of redundant computations and the number of instructions executed.

Constant propagation (CP) involves replacing variables or expressions with their known constant values at compile time. This optimization can improve the performance of a program by reducing the number of instructions executed.

1.4 Back-end

The back-end's task is to translate the result from the Optimizer into a target-specific language, generally into the target-specified instruction set architecture (ISA). So the instructions represented in IR should be rewritten into a sequence of the target instructions.

After producing the target code, there can also be the last part that does the target-dependent optimization because translation often leaves bits that can be further optimized. Such Optimizer can be, for example, *Peepholer*. Peepholer looks at a few instructions in the 'peephole' and tries to find a pattern to be optimized within the peephole. The peephole is shifted throughout the program.

1.4.1 Instruction Selection

Instruction selection maps IR instruction to target instruction so that the result is efficient in execution but requires the minimum amount of instructions and target resources possible to achieve the correct result. Generally, it is responsible for converting the constructs in IR into machine code. How much this process is demanding depends on the desired quality of the generated code, the level of IR, and the target instruction set architecture.

The process involves analyzing the intermediate representation of the program and selecting the most efficient sequence of machine instructions to implement each operation. The process can be described in several steps:

- Generating a candidate set of instructions: The compiler must generate a set of possible machine instructions for each high-level construct to implement the operation.
- Evaluating the candidate set: The compiler must evaluate each candidate instruction's correctness, efficiency, and adherence to the target architecture's instruction set. This evaluation involves analyzing the instruction's latency, throughput, register usage, and other factors impacting the program's performance.
- Selecting the optimal instruction: Based on the evaluation, the compiler selects the most efficient instruction from the candidate set to implement the operation. This selection process may involve heuristics, such as choosing the instruction with the shortest latency or the lowest register usage.

Various techniques can be used for the generation of the candidate set and selection of the optimal instruction:

- *Table-based code generation:* This technique uses tables or matrices to map high-level language constructs or IR code to corresponding or machine instructions. These tables can be hand-coded or automatically generated by a tool.
- *Pattern matching:* This technique involves matching patterns in the high-level language construct or IR code with pre-defined patterns of code or machine instructions. These patterns can be hand-coded or automatically generated by a tool as well.
- *Tree-based code generation:* This technique involves representing the high-level language construct or IR code as a tree and using tree traversal algorithms to generate the corresponding or machine instructions.
- *Dynamic programming:* The problem is formulated as an optimization problem, and an optimal solution is found by dynamic programming algorithms.
- *Constraint solving:* The problem is formulated as a constraint satisfaction problem, using constraint-solving algorithms to find a solution that satisfies the given constraints.

More straightforward techniques considering only instruction at a time generate inefficient code; therefore, a peephole optimization often runs after the IS. The next section 1.4.2 describes the peephole optimization. Chosen techniques overall depend on various factors such as the complexity, form of the IR, the performance requirements, and the available resources.

1.4.2 Target specific optimizations

Target-specific optimizations are compiler optimizations that are designed to take advantage of the specific features and capabilities of a particular hardware architecture or target platform. Peephole optimization is a type of local target-specific optimization technique, which improves the performance of generated code. It involves analyzing a small window of instructions in the code, known as a 'peephole,' and replacing the instructions with a more efficient sequence of instructions that achieves the same result.

The peephole optimization technique is based on the idea that small code sequences often contain inefficiencies that can be removed by analyzing the instructions in a small window. The window typically contains three to five instructions, and the optimization is applied to all possible windows in the code.

1.4.3 Register Allocation

Register allocation (RA) is a technique that involves mapping the values of a program to the limited number of registers available in the target, in order to minimize the number of memory accesses required during program execution. By keeping frequently used variables in registers, this technique can improve program performance.

The problem of register allocation is complex and falls under the category of NP-hard problems. Therefore, finding the optimal solution to register allocation is not feasible in practice. However, there are several simple solutions to this problem that can be used in compilers:

- *Naive approach:* This approach simply allocates a register for each variable used in the program. When no registers are full, pick the oldest for release. However, this approach may lead to many register spills (i.e., storing the register value back to memory), and hence performance may not improve.

- *Graph coloring approach:* This approach views the register allocation problem as a graph coloring problem, where the nodes represent the variable's live ranges and the edges represent conflicts between them. This approach can be applied to both local and global register allocation problems.

- *Linear scan approach:* This approach uses a linear scan of the program to allocate registers to variables. This simple and efficient approach may not always generate an optimal result.
- *Heuristic-based approach:* This approach involves using heuristics to guide the register allocation process. Examples of heuristics include the use of spill costs and register pressure.
- *Hybrid approach:* This approach combines multiple register allocation techniques, such as graph coloring and linear scan, to achieve better performance.

The peephole optimization should be performed again to obtain even better results because RA adds the spill code, which can also be optimized.

The register allocator's tasks are:

- To maintain all registers for values that reside in registers.
- To maintain a spill table – mapping all values in memory and their addresses.
- To decide which values in registers will be spilled to free up space for new values.
- To resolve at which point of execution a restoration of spilled value is required. That means where to load value back to register to keep the values in registers consistent with the computation that requires these values.

Coloring Allocation

The Coloring Allocation's task is to select which target registers will carry which values and which values to spill to minimize the number of spills over the execution. It was first introduced by Chaitin et al.[3]

The idea is to represent this problem as a graph in which nodes represent live ranges (temporary values, variables) that need to have assigned a target register and edges connect interfering live ranges, i.e., live ranges alive at the same time, which might be conflicting for a target register. Register allocation this way is reduced to a graph coloring problem: which colors (registers) will be assigned to the nodes such that two nodes connected by an edge do not share the same color. The graph can be built using a liveness analysis.

The allocator consists of the phases shown in Figure 1.6 are following:

- *Renumber:* Detect live ranges and number them in the IR.
- *Build:* Compose the interference graph of the live ranges.

- *Coalesce:* Merge the live ranges of variables with the same value – i.e., those created with copy instructions and rebuild the graph.
- *Spill cost:* Determine the cost of spills for each variable.
- *Simplify:* Color the graph nodes and continue with selection.
- *Spill Code:* Insert spill code, including load, stores, or move instructions.
- *Select:* Select a register for each position (instruction) that requires the register; if not possible, add a spill code.

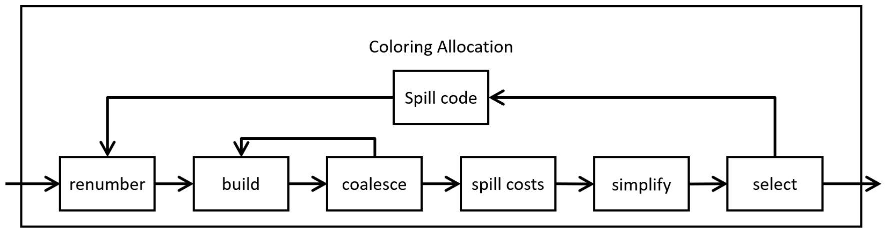


Figure 1.6: Chaitin et al.'s iterative graph coloring based register allocator

1.4.4 Instruction Reordering

Instruction reordering is useful for superscalar processors when more instructions without specific dependencies can be executed together. Superscalar processors use instruction parallelism to achieve faster execution. Each instruction is divided into some execution parts, such as instruction fetch, operand fetch, ALU, and others. However, problems called hazards appear when the processor wants to start the execution of an instruction that does not have a result ready while it is still in the process of computation. That way, the processor needs to insert *NOP instruction* and therefore stall in computation. So the task is to prepare the code for the processor in a way that the instructions do not need to wait on each other and utilize all parts of the processor as much as possible therefore if it might be helpful to reorder instructions to reduce the time necessary for a program to finish.

Nowadays this problem is solved by the target itself. The target has buffers and computes instructions that are eligible for computation in a different order. The eligible instructions are those that will not produce a different result when executed now (out of order). This way compiler reordering task is not as significant as it was before.

1.5 Case Studies

It is reasonable to talk only about two compiler suites at this time. Nowadays, two main compiler suites exist: The GNU C Compiler (GCC) and the LLVM. There also exist other compilers like javac, D compiler, etc. However, these are language or target specific; therefore, the inner structure differs from the modular, in which this thesis is interested.

1.5.1 GCC

GCC (GNU Compiler Collection) is a compiler suite developed as a GNU project. Richard Stallman founded the project to create a completely free operating Unix-like system. However, no free compilers were available for C language he could use by then. So he decided to develop his own.

The first version of GCC was released in 1987 under the name GNU C Compiler. This release significantly impacted the development of openly licensed software because even nowadays, C language is widely used.

In 1997 a fork called EGCS was founded. This project was better at optimization and had better support for C++ and Fortran. This project was an answer to a too-strict policy for upgrades. After two years, the Free software foundation acknowledged project EGCS as an official compiler of the GNU project. After that, these projects were united again.[4]

GCC supported many different target architectures, from micro-controllers to the most performant ones. Also, it can translate many languages, including C, C++, Fortran, Java, or Go, and is used as the default translator in modern Unix systems — in many different variants, GNU/Linux or BSD.[4]

Intermediate representations

GCC uses all together three forms of internal representation GENERIC, GIMPLE, and RTL. GENERIC serves as an independent representation of the higher-level input language. This form is used in the front-end. It has a form of a typed AST. Types are common for all GCC front-ends. It is also possible to define own types of AST nodes if the source language requires so.

```c
int g = 4;
int main(){
    int l = 16;
    return g < l;
}
```

Listing 1.3: Example C code

After the input is unified, the GENERIC is translated into GIMPLE using so-called *gimplifier*. If own types were defined, functions translating own types to GENERIC or straight to GIMPLE are required. All platform-independent optimizations are then made on GIMPLE, which is a subset of the generic

```
main ()
{
  int l;
  int D.2318;

  l = 16;
  g.0_1 = g;
  _2 = l > g.0_1;
  D.2318 = (int) _2;
  goto <D.2319>;
  D.2318 = 0;
  goto <D.2319>;
  <D.2319>:
  return D.2318;
}
```

Listing 1.4: Example GIMPLE for C code from example 1.3

language. Complicated structures that are possible to represent in GENERIC are forbidden in GIMPLE and therefore disassembled into less complicated constructs with only three operands. This way, a three-address code is obtained. Gimple is also transformed into an SSA form. In the documentation, they call it 'Tree SSA.' The last transformation step is to convert 'HIGH GIMPLE' to 'LOW GIMPLE.' The difference is a description of control flow. The 'HIGH GIMPLE' contains control-flow structures, but the 'LOW GIMPLE' contains only conditional jumps.

```
main ()
{
  int l;
  int D.2318;
  int g.0_1;
  _Bool _2;
  int _5;

  <bb 2> :
  l_3 = 16;
  g.0_1 = g;
  _2 = l_3 > g.0_1;
  _5 = (int) _2;

  <bb 3> :
<L0>:
  return _5;

}
```

Listing 1.5: Low GIMPLE representation for C code from Example 1.3

The final form is named Register Transfer Language, known as RTL, which in the back-end serves as an independent interface for performing back-end-dependent optimizations. RTL is inspired by Lisp lists and is composed of 'RTL Classes,' which can be combined into more complex ones called 'RTL expressions.' These expressions are used to precisely describe what a target instruction does, how it behaves, which types it uses, and other properties. Example 1.6 shows commented the RTL representation.

Optimizations

Optimization in GCC is performed throughout the compilation in several parts. Some are target-independent, others target-dependent. The Target dependent optimizations are performed in the back-end on the RTL form. The first optimization is performed while conversion into GIMPLE is made; the part is described as Analysis and Optimization of GIMPLE tuples. This part converts The Generic into SSA, and alias analysis is also performed.[5] The second important construct for optimization and analysis is Control Flow Graph (CFG). It is built on top of the intermediate representation. The demanding part of CFG is maintaining it throughout the compilation and optimization, as reconstruction is too expensive, and some information may not be obtainable again. CFG keeps the information about Basic Blocks and profile information like frequency and counts, specifying how often the part is executed. These statistics are necessary to decide which optimization should be used because it might trade between code size and execution time.

There is also a component in GCC called Link Time Optimization (LTO) which works by dumping GIMPLE or RTL into a file and then traversing the file looking for parts that can be optimized. This approach is beneficial because it has all the module's information accessible at link time.

The inter-procedural optimizations (IPA) are performed and organized in IPA passes in the Link Time Optimization (LTO) section. The IPA passes are organized into three categories — small, regular, and late passes. Each pass has several stages, and depending on the category, they are run in a different part of the compilation — operating at different times of the compilation. Stages are LGEN, WPA, and LTRANS. These stages describe the general needs for general optimization:

- Analysis which is made in *LGEN time* generates and writes down a summary.

- Propagation or execution that transforms a copy of CFG is described as *WPA time.* This way, the original functions, and variables stay untouched because there would be inconsistency between starting analysis and operations that each pass wants to make. This process may create conflicts. At this time a decision on how to resolve these conflicts that may some passes create is made as well. For this operation are useful profile options as well.

```
(note 1 0 3 NOTE_INSN_DELETED)
(note 3 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(note 2 3 5 2 NOTE_INSN_FUNCTION_BEG)
(insn 5 2 6 2  // l_3 = 16;
  (set
    (mem/c:SI
      (plus:DI
        (reg/f:DI 77 virtual-stack-vars)
        (const_int -4 )))
    (const_int 16 ))  -1
     (nil))
(insn 6 5 7 2  // g.0_1 = g;
  (set
    (reg:SI 82 [ g.0_1 ])
    (mem/c:SI (symbol_ref:DI ("g") [flags 0x2]
            <var_decl 0x7f9876c8ee10 g>) [1 g+0 S4 A32])) -1
     (nil))
(insn 7 6 8 2  // _2 = l_3 > g.0_1;
  (set
    (reg:CCGC 17 flags)
    (compare:CCGC
      (mem/c:SI
        (plus:DI
          (reg/f:DI 77 virtual-stack-vars)
          (const_int 4 -1 )) )
      (reg:SI 82 [ g.0_1 ]))) -1
     (nil)
(insn 8 7 9 2   // (int) _2
  (set
    (reg:QI 83 [ _2 ])
    (gt:QI
      (reg:CCGC 17 flags)
      (const_int 0 ))) -1
     (nil))
(insn 9 8 12 2    // _5 = _2;
  (set
    (reg:SI 84 [ _5 ])
    (zero_extend:SI (reg:QI 83 [ _2 ]))) -1
     (nil))
(insn 12 9 16 2   // return _5;
  (set
    (reg:SI 85 [ <retval> ])
    (reg:SI 84 [ _5 ])) -1
     (nil))
(insn 16 12 17 2
  (set
    (reg/i:SI 0 ax)
    (reg:SI 85 [ <retval> ])) -1
     (nil))
(insn 17 16 0 2
  (use (reg/i:SI 0 ax)) -1
     (nil))
```

Listing 1.6: Exmaple RTL for C code from Example 1.3

- Transformation which sees the result of *WPA time* and changes the original structures underneath by the result obtained in *WPA time*. This stage is referred as to *LTRANS time*.

The small IPA is compact because it cannot be separated into stages. Examples of small IPA are 'IPA remove symbols,' which performs reachability analysis and removes unused symbols, or 'IPA increase alignment,' which is helpful for vectorization.

Regular uses all stages like 'IPA constant propagation,' 'IPA inline,' or 'IPA identical code folding.'

Lastly, the late IPA is described as performing all at once as small but later after regular passes are performed. To late IPA's belong, for example, 'IPA points-to analysis.'

GCC has a WHOPR option that performs optimizations in parallel, that is the reason why the passes are designed like this, especially in *WPA time* where it operates on the copy.

Back-end

The GCC back-end uses RTL and definition files of the target instruction set. The task is to transform the RTL to a shape that is similar to symbolic address language. So the translation to a symbolic address language can be trivially performed. The work of the back-end can be separated into several steps

RTL generation also called 'expand'. This first part converts GIMPLE to RTL with the help of instruction patterns. Some mandatory definitions exist in the back-end, but also optional can be implemented. If there are none, the compilation ends with an internal error.

optimization After RTL is generated, some optimization steps are run, including condition conversion, tail call elimination, or common sub-expression elimination (CSE).

instruction combining Similar to peephole optimization tries to combine several RTL instructions into one. Together with previous steps, this belongs to instruction selection and has the greatest impact on which target instructions will be in the final product.

register allocation This part contains several passes over the RTL, continuously replacing virtual registers called 'pseudo registers' with fixed ones and replacing instruction patterns. Hence, the arguments are acceptable by the target definitions. Register spilling and restoring and function prologs and epilogues — etc. calling conventions — all these steps are made in this part too.

final the last pass that writes in the output file. For every RTL instruction, an adequate representation in symbolic address language is selected with the help of the target description.

Target description is done with RTL and C macros. The description of the general properties and characteristics of the target architecture is a composite of three files one C-like pair *target.h*, which defines the target properties, and *target.c*, which defines an extra functionality which overrides the defaults, and one *target.md* which serves as an input to the compiler in compile time.

1.5.1.1 Conclusion

In summary, GCC is a mature and reliable compiler suite that has been used for decades to build high-performance software. Its flexibility, performance, and open-source nature make it a popular choice for developers around the world.

1.5.2 LLVM

The LLVM Project is a collection of compilers and toolchain technologies — programs and libraries written in C++.[6]

The project was founded in 2000 by a research group of developers at the University of Illinois in Urbana Champaign. The original goal of this group was to examine dynamic translation techniques. Therefore the name was Low-Level Virtual Machine, but nowadays, the original was extended further. It is used in many projects, so it changed the name to LLVM as the last name's abbreviation since the original idea drifted slightly in a different direction.[7]

Essential parts of LLVM are platform independent optimizer, code generator of LLVM Core, and compiler of C, C++, and C-Objective language called Clang. However, this project supports more languages like Fortran, Ada, Rust, Swift, etc. Some of them belong to the 'DragonEgg' project, which combines the GCC front-end with the LLVM optimizer and back-end. The following useful thing is that LLVM contains other tools like a debugger, assembler, or even a standard C++ library. That is why LLVM began to be used with branches of UNIX systems as an alternative to GCC and GNU Binutils. It is composed of modules and libraries useful for both programmers and end-users.

The LLVM Core is the core part of LLVM. Most of the time, by referring to LLVM, it is referred to this part. It consists of libraries and programs working with intermediate representations called LLVM IR. LLVM developers use these to interact with the IR; end-users should not need to use them themselves. A description of such programs follows:

llvm-as Is a program that converts assembler LLVM IR from a text file to the program's binary representation. The instructions in bytecode format are still from the LLVM IR, not the native bytecode ones.

llvm-dis The program is used to convert bytecode byte form into LLVM IR text form.

llvm-link Servers as Linker for LLVM bytecode. Combines more binary files containing bytecode into one output file.

lli The program for interpreting the LLVM bytecode. Runnable on x86 architecture, Sparc, and PowerPC and supports just-in-time (JIT) translation as well.

llc This program serves as the back-end of LLVM. Its task is to translate LLVM bytecode (IR) into instructions the target supports. The output can also be in the language of symbolic addresses or an object file. This program is a cross-translator as it can translate into different targets without the need for the recompilation of the whole project.

opt Is the middle-end of LLVM. It contains optimization techniques that LLVM does. From input reads in LLVM bytecode or text LLVM IR form. Optimizes it and optimized code saves into output in LLVM IR.

Bitcode

LLVM IR is a universal SSA form represented by a three-address code that creates the interface between the front-end where it is created, the middle-end where optimization passes use it, and the back-end where it is transformed into target code. The best property of LLVM IR is its independency on all inputs and its 'storability.' Hence, it is possible to store it anytime, load it afterward, and continue without any information loss. At the end of the translation ending with the same result as no memory dump was performed. This property is very useful for scaling for parallel runs. Programs written in the form of LLVM IR are composed of modules where each module refers to each input file. Each module specifies a symbol table, global variables, and functions of the input program.

Instructions are divided into several categories

- *terminating instrucitons* are those that do not produce any computation result; therefore, its return type is typically *void*. They are used to alter the control flow of a program. Instructions like *indirectbr*, *br*, or *ret* belongs here.
- *binary operations* represent the arithmetic operations, take two input arguments of the same type, and produce one value of the same type as input ones. For example *div*, *mull*, *add*.
- *memory instructions* represents work with memory. The most used are *load* and *store*, but instructions like *alloca*, which allocates memory on

```
@g = global i32 4, align 4

define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 16, i32* %2, align 4
  %3 = load i32, i32* @g, align 4
  %4 = load i32, i32* %2, align 4
  %5 = icmp slt i32 %3, %4
  %6 = zext i1 %5 to i32
  ret i32 %6
}
```

Listing 1.8: Example of LLVM IR

the stack, belong here too, as well as *cmpxchg* or *atomicrmw* representing atomic operations with memory.

- *And others* like unary, conversion, vector, aggregate, or unclassified operations such as *call*, *'trunc .. to'* instructions.

The LLVM type-system contains types for

- plain data types like integer numbers and floating point numbers,
- functions, which define types of arguments and type of the return value
- special complex types for representing structures and arrays.

```
int g = 4;
int main(){
    int l = 16;
    return g < l;
}
```

Listing 1.7: Example C code

The naming convention for variables takes the original variable's name from the source program. If the variable is compiler-defined, it is named by a number. Moreover, all variables have a prefix defining if it is a local variable prefixed with the '%' symbol or a global variable prefixed with the '@' symbol. If the LLVM IR instruction produces a result, it is stored in a variable.

Pass Scheduler

The Pass Scheduler is at the heart of the LLVM Optimizing section. Its task is to schedule, organize and manage passes over the LLVM IR that performs optimization. Moreover, the goal is to be as effective as possible while minimizing the compilation time. The Pass Scheduler is divided into components:

- Pass Manager which is responsible for organizing the passes and scheduling their execution. The LLVM supports multiple types of Pass Managers like FunctionPassManager or CGSCCPassManager (abbreviation for Call Graph strongly connected component).
- Pass Groups is a collection of passes that are meant to be executed together. This way, it is possible to define the order in which the passes inside the same group will be executed. Optionally, criteria can be specified to enable or disable the execution of the passes.
- Pass Ordering defines a process determining the order in which passes and groups will be performed. Various techniques are used to determine the optimal pass order. These techniques decide with the help of profiling data, user hints, and heuristics.
- Pass Dependencies helps to specify the relations and relationship between passes, mainly to determine the correct ordering.
- Pass Manager APIs specify the interface for managing pass groups and passes. Management concerns adding, removing, configuring, and getting information about dependencies and ordering.

Back-end

The main part of the LLVM back-end is the target-independent code generator. The code generator is composed of six components[8]

Target-independent algorithms which are the core of the platform-independent code generator. There are implemented in separate phases of code generation like register allocation, instruction selection, or machine code optimizations.

Abstract target description These implementations describe the interfaces necessary for the core to be able to produce code for the target.

Implementations of the abstract target. Machine descriptions that use the LLVM components and can optionally provide target-specific passes to complete a code generator for a specific target.

Classes for representation of generated code for the target. In this part, a representation of the target code is present but abstract enough to be shared with all targets. At this level, there are still concepts like ‘jump table’.

Machine code. The representation is very similar to a form of an object file. This representation is called the MC layer because all high-level information is ignored, and things like “jump tables” no longer exist.

The MC layer represents all the low-level detail like labels, sections, and assembler instructions.

The target-independent JIT components. Completely target independent, uses its definitions and structures for the target description.

The process inside the code generator itself can be divided into parts that follow each other. From a high-level perspective, the code generator parts are the following:

Instruction selection In this stage, an efficient way of expressing the LLVM program is conducted. The IS assigns information about virtual registers and adds physical registers in cases enforced by constraints like calling conventions or target specifications. At the end of this step, the representation continues in the form of a DAG that contains target instructions.

Scheduling and Formation At this stage, instructions in the dag are reordered and linearized back to meet the demands of efficient execution at the target.

SSA Optimizations This pass is optional and provides an option to realize optimizations on the linearized form of the IR instructions.

Register Allocation The pass generates an extra 'spill' code and assigns virtual registers of the target machine. After this pass is finished, virtual registers are replaced with concrete target registers; therefore, after this step, the SSA rules are violated, and the program is no longer in SSA form.

Prolog/Epilogue Code Insertion After generating machine code, additional info is added. Take a function and calling conventions as an example, a function needs to operate with the stack, and the stack offsets are known now, so the placeholders are replaced with available sizes. Additionally, optimizations, like frame-pointer elimination[3] and stack packing, are performed here too.

Late Machine Code Optimizations The last finishing touches that are required, for example, peephole optimization or scheduling of the spill code.

Code Emission This last phase produces a representation in an MC layer where it is prepared to dump all the results into a file in an assembler form.

[3]Frame pointer is usually a dedicated register for access of local variables and arguments, eliminating complex access means overall faster memory access.

1.5.2.1 Conclusion

In summary, LLVM is a powerful and flexible compiler infrastructure that supports multiple programming languages and provides a range of optimization options to improve code performance. Its modular design and support for JIT compilation make it a popular choice for developers who require a modern and customizable compiler infrastructure.

CHAPTER 2

Design and Implementation

The design of this compiler is modular in all three parts: front-end, Optimizer, and back-end.

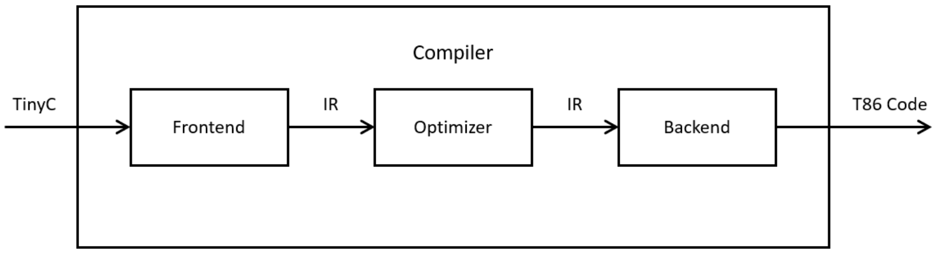


Figure 2.1: Compiler

The Figure 2.1 shows that the compiler uses the same design as modern compilers from a high-level perspective. The *front-end* has the same task to convert an input program, in our case written in TinyC, to an Internal representation. The *Optimizer's* task is to make the program 'better' mainly in terms of performance but preserve the correctness of the program. It should be possible to omit the part with the *Optimizer*, and the compiler should still produce correct, functional code. The last part is *back-end.* Its role is to translate intermediate representation to the target program; in our case, the target program comprises target instruction specified by the t86 target machine.

In Figure 2.1, there is also noticeable that the program changes representations. Chronologically listed representations are:

- TinyC,
- then abstract syntax tree (AST), which is hidden inside the front-end part,
- then intermediate representation (IR),
- and then the set of t86 target instructions (mainly referred to as Assembler).

Each part of the compiler works with one representation or translates one to another. Each representation is better for different purposes; for example, AST is useful for Semantic analysis because it is a high-level representation, but it is not so convenient when deciding which sequence of target instruction needs to be used to produce an optimal result for this purpose a low-level representation is more practical.

The interesting part of the front-end is the translation to IR, which is introduced in the next section.

2.1 Intermediate Representation

The intermediate representation follows the register IR principle. Every IR instruction itself also represents a value that it returns. The values will eventually be stored in registers or memory. Registers are of two types: Integer or Float registers. In IR, we can assume that we have infinitely many registers. So every instruction has its own register, even if it is of type void, representing no value.

Instructions are based on LLVM's IR[9] with slight differences in instructions like ArgAddr, which represents a simplified way to get an address of some argument in a function. Allocations are separated into more instructions than just one to simplify the approach and implementation between local and global scope allocation.

The whole IR is not just the specification of the instructions that can be found in the chapter C in the appendix of this thesis. Instructions are kept in Basic Blocks. The Basic Block represents a sequence of instructions where execution goes linearly. After executing the first instruction of the Basic Block, all other instructions are executed in the order of the sequence as well, right after the first one. From that following property is derived: Basic Block is terminated by special instructions called terminators. Terminator instruction is, for example, Jump instruction. Furthermore, the Jump instructions only can land at the first instruction of the targeted Basic Block.

The next part that wasn't mentioned yet is how functions are represented. Functions consist of a sequence of Basic Blocks. The starting point of a function is the same as for the first Basic Block of this function.

The whole representation of an input program in IR is following: The *Program* contains *Functions*, *string literals* used in the program, and one extra Basic Block which contains global scope expressions – code that will be executed at the beginning before the start of the main function.

Let us take a closer look at the information that Instructions hold. These data are necessary for instruction selection (introduced in the back-end) such as:

- which type of register has to be used for the result of this instruction
- instructions from which to take results as input
- value type of instruction that came from type checker

Type information is required because some instructions, such as Load, can work with either integer or float representations. The only information on how the result is represented in memory comes from this type checker.

The other essential parts of IR are types. TinyC has 4 'Plain old data' (POD) types Int, Double, Char, and Void. Boolean is not an explicit type, but it is necessary for working with conditions, so technically it is substituted by Int. As mentioned earlier, only two types of registers are available in IR. Here comes the first simplification: Double translates to Float registers and the rest (Int, Char) into Integer registers. Integer registers can also carry addresses known as Pointers in TinyC. Explicitly described like this:

- *Int, Char* specifies that Integer based register needs to be used to store the result of this instruction.
- *Double* specifies that Float based register needs to be used to store the result of this instruction.
- *Void* indicates that this instruction does not have an output that can be used. For example, Jump instruction does not represent a value in the program.

In TinyC, there are a few more types. The missing are:

- *Alias* is another representation of existing type. The Alias type specifies only a base type that it stands for.
- *Pointer* The Pointer type specifies a base type that points to another type that specifies the Pointer type. It fits into an integer register.

- *Array* The Array type specifies, apart from the base type, the size of how many values the array holds. This type does not fit into a register, and the simplification that every type fits into a memory cell does not apply to this type.
- *Struct* The Structure type specifies a mapping from a Symbol (which stands for a member name) to a type. Struct, generally, does not fit into a register either.
- *Function* The Function type specifies a type returned from the function after it is executed and a sequence of types representing types of function arguments.

They differ in some properties. They depend on other types — they use them for their own correct definition. Pointer and Structure can be recursive. Structures no longer follow the simplification that all variables fit in one memory cell or just one register.

There are three types used in IR:

- Integer
- Float
- Void

The 'Void' type is just a placeholder indicating no value.

The reason for having only two register types in IR is that the target only supports these two types. They significantly differ in representation; therefore, specialized instructions are introduced in the target ISA. Adding two float instructions requires an extra instruction than adding two integers. Also, working with more types of specialized registers would translate into the same instructions; therefore, it is easier to merge them because representing more types of registers does not raise any interesting problems.

2.2 AST to IR Translation

The first step of translation is to convert the TinyC language to AST. That is done in Lexer and Parser located in package tinyC. That was done for us and now it is the time to convert AST to intermediate representation (IR or sometimes addressed as Intermediate Language IR).

Converting AST to IR is done with the help of the Visitor pattern. In this work, a visitor pattern is used like this. Each node of AST has an assigned rule on converting that AST node into an IR Instruction. Some IR Instructions take arguments; these arguments are generated by AST traversal, by pair visit and accept functions from the visitor. After completing the top-down traversal, one or more IR instructions are formed and saved into adequate

```
//TinyC                        //IR
int main(){                    function:
    return 4 + 5; //!9         main:
}                                  bb_0:
                                       r0: int = LoadImm 4
                                       r1: int = LoadImm 5
                                       r2: int = ADD r1: int r2: int
                                       Return r2: int
```

Listing 2.1: Example of translation of plus operator from TinyC language to IR.

Basic Blocks. This visitor is run inside a module IRBuilder, which manages the context of the Program and sews created Instructions together into Basic Blocks, the Basic Blocks to Functions creating the whole desired product. Let us see some examples of how it looks in practice.

When creating the first version of IR representation, working with IR registers is simple; whenever a new register is required, take an unused one, continuing to infinity.

In Figure 2.1, there is a simple example of a translation that a calculator can also do — some Binary operations. Compiling Unary instruction is the same, but naturally with one less operand. In order to compile the return statement in the example, the result from the binary operation was required. However, to get the result of a binary operator, the results of the left and right sides need to be compiled. There are just a number literals, so there the AST traversal ends.

Control Flow expressions

Let us move to a part that separates computers and calculators. That is deciding what to do next. In our terminology, conditional jumps. TinyC language represents it as if statements, switch statements, and loops like `for`, `while`, and `do-while`.

Jump instructions can jump only to the beginning of Basic Block — to its first instruction. The second important fact is that every Basic Block ends with a 'terminator' instruction. When the 'terminator' instruction is present in Basic Block, no other instruction can be added to this Basic Block; it is classified as closed. Terminator instruction changes the control flow of a program like previously mentioned `Jump` or `CondJump` instructions.

In Example 2.2, you can see how to translate the if statement and Basic Blocks in action.

Analogous to the examples before is the translation of loops. Similar to the condition translation with an extra jump back after the body of the loop to the loop condition.

```
//TinyC               //IR
                      bb_0:    r0: int = LoadImm 0
if(0){                         CondJump r0: int bb_2 bb_1
                      bb_1:    r2: int = LoadImm 88
    print('X');                PutChar r2: int
                               Jump bb_3
}else{                bb_2:    r5: int = LoadImm 48
    print('0');                PutChar r5: int
                               Jump bb_3
}                     bb_3:
...                            ...
```

Listing 2.2: Example of translation of an if statement from TinyC language to IR.

```
// TinyC                // IR
                        bb_0:
int a = 0;                  r0: int = LoadImm 0
                            r1: int = AllocL 1 (size: 1)
                            Store [r1] r0: int
                            Jump bb_1
while(a<5){             bb_1:
                            r4: int = Load [r1]
                            r5: int = LoadImm 5
                            r6: int = Lt r4: int r5: int
                            CondJump r6: int bb_3 bb_2
                        bb_2:
    a++;                    r8: int = Load [r1]
                            r9: int = Copy r8: int
                            r10: int = Inc  r8: int
                            Store [r1] r10: int
                            Jump bb_1
}                       bb_3:
...                         ...
```

Listing 2.3: Example of translation of a while statement from TinyC language to IR.

In Figure 2.3, you can see that translation of the while cycle needs to create a total of three Basic Blocks, the first for the loop condition, the second for the loop body, and the last for the part that follows after the loop.

The following example concerning the control flow belongs to the switch, which requires a bit of thinking while solving as an if-else branching as in this work. Generally, in other compilers, a jump table is created in memory, offsets are counted, and the address where to jump is obtained from that table. That is quite a complicated process. The less complicated path was chosen, sacrificing a small amount of effectiveness.

Compiling a switch with an if-else chaining can still be demanding when fallthrough functionality is present with the default case that stands somehow aside from other cases and its order but not really while we need to know the order of each of the cases. The translation is shown in the Example 2.4.

To summarize, when translating a switch statement, there are a few tricky parts:

The first part is to connect Basic Block that must be created. There are two types of Basic Blocks: first for condition and second for the body of the case.

Second part is handling the default case, which can be in the middle, as a first case, as a last case, or missing entirely, and correctly patching the jump instructions between the Basic Blocks.

Last part is to handle the break statement inside the switch. Therefore, a new context is created, extending the old context, and can answer with the correct Basic Block to jump at whenever necessary. After leaving the part where the switch is translated, the context needs to be marked as left. That is necessary because in case this is not done situation with nested switches would not work correctly, and in the outer switch case, the jump created by breaking would jump on the last (inner) switch compiled, thus creating a loop.

The last part of creating context clarifies the reason why the Basic Blocks are ordered in a counter-intuitive way.

The `after_bb` needs to be prepared while the context is created, so sometimes the Basic Block is created in weird positions other than expected.

The last part of control flow examples belongs to the Logic Binary Operators ('&&' and '||'). If it is unclear, let us elaborate. Short circuit evaluation is the term that explains why. Short circuit evaluation, in any programming language, is an expression `x and y` is equivalent to the conditional expression `if x then y else x`, and the expression `x or y` is equivalent to `if x then x else y`. In either case, `x` is only evaluated once.

The generalized definition above accommodates loosely typed languages with more than the two truth values `True` and `False`, where short-circuit

```
// TinyC                // IR
                         bb_0:
int a = 0;                   r0: int = LoadImm 0
                             r1: int = AllocL 1 (size: 1)
                             Store [r1] r0: int
char c = 'A';                r3: int = LoadImm 65
                             r4: int = AllocL 1 (size: 1)
                             Store [r4] r3: int
switch(c){                   r6: int = Load [r4]
                             Jump bb_case_65
                         bb_after:
                             ...
case 65:                 bb_case_65:
                             r8: int = LoadImm 65
                             r9: int = Copy r6: int
                             r10: int = Neq r9: int r8: int
                             CondJump r10: int bb_succ_65 bb_case_66
                         bb_succ_65:
  a = 1;                     r12: int = LoadImm 1
                             Store [r1] r12: int
                             Jump bb_succ_66
case 66:                 bb_case_66:
                             r15: int = LoadImm 66
                             r16: int = Copy r6: int
                             r17: int = Neq r16: int r15: int
                             CondJump r17: int bb_succ_66 bb_case_0
                         bb_succ_66:
  a = a + 1;                 r19: int = Load [r1]
                             r20: int = LoadImm 1
                             r21: int = Add r19: int r20: int
                             Store [r1] r21: int
  break;                     Jump bb_after
                         bb_case_0:
                             Jump bb_default
default:                 bb_default:
  a = 4;                     r26: int = LoadImm 4
                             Store [r1] r26: int
  break;                     Jump bb_after
}                        bb_8:
...                          Jump bb_default
```

Listing 2.4: Example of translation of a switch statement from TinyC language to IR.

```
// TinyC               // IR
int main(){            function:
                       main:
                         bb_0:
    int a;                 r0: int = AllocL 1 (size: 1)
    a = 15;                r1: int = LoadImm 15
                           Store [r0] r1: int
                           r3: int = Load [r0]
    int b = a < 30         r4: int = LoadImm 30
                           r5: int = Lt r3: int r4: int
        &&                 CondJump r5: int bb_2 bb_1

                         bb_1:
         (a = 30);         r7: int = LoadImm 30
                           Store [r0] r7: int
                           CondJump r7: int bb_2 bb_3
                         bb_2:
                           r10: int = LoadImm 0
                           Jump bb_4

                         bb_3:
                           r12: int = LoadImm 1
                           Jump bb_4

                         bb_4:
                           r14: int = Phi
                           r12: int <--bb_3,
                           r10: int <--bb_2
                           r15: int = AllocL 1 (size: 1)
                           Store [r15] r14: int
    return b;              r17: int = Load [r15]
}                          Return r17: int
```

Listing 2.5: Example of translation of '&&' Operator from TinyC language to IR.

operators may return the last evaluated subexpression. This process is called the 'last value.' For a strictly-typed language, the expression is simplified to `if x then y else false` and `if x then true else y` respectively for the boolean case.[10]

In our case, these operators return binary values 0 or 1. However, when we take short-circuiting in play, it depends if the second operand is evaluated, so conditional jumps and Basic Blocks must be created. Therefore it belongs to this section.

Translation to IR is, therefore, tricky because it creates four Basic Blocks for branches lhs evaluates to true/false and for the whole operator when ending true or false, and then after for joining the control flow after this operator the last Basic Block. Every Basic Block needs to be filled with correct instructions.

In Example 2.5, you can also see the usage of the ‘Phi’ instruction from the SSA property. Registers in this IR keep SSA property. SSA is a property of intermediate representation requiring each variable, or register in our case, to be assigned precisely once and defined before it is used. This is not the case here, as the register for storing value 0 and the register for storing value 1 cannot be the same.

In the case of the Logic Binary operator, this could be solved using a memory that is not SSA. Otherwise, it would be necessary to store a temporary value and then load it, which would be time inefficient.

Memory Management expressions

The next part of translation belongs to memory, variables, and values that the compiler needs to work with.

When memory is mentioned, most students recall the *new* operator from C++ and heap with its allocation. That is *not* the analogy we want to work with; we do not work with dynamic allocation, to be clear, because the heap is a runtime concept. That means only static size variables and values are considered in this thesis.

In order to save a value to the memory, we need to tell it to give us an address where we can store the value. This transaction is done with the help of Alloc Instructions. The Alloc instructions differ in context if it is AllocL or AllocG. AllocL represents allocation inside a function and AllocG outside. AllocL instruction allocates a place of specific memory size and represents a particular variable’s address. A stack is used for local allocation, so AllocL instruction asks the stack for an address. In our case, a simplification is present since the t86 target’s approach to memory works only with cells instead of bytes. Therefore allocation of a basic type takes one cell. So we depend on counting memory cells instead of bits and bytes.

With the help of Alloc instruction, we can represent an address in memory. To be able to save and restore values to and from memory, a Store and Load instructions, respectively, are required.

Store and Load instructions require the address and the value that needs to be stored in memory at the address (resp. loaded from memory at the address). The address needs to be found somewhere as well. For this purpose, IRBuilder has a table, the so-called symbol table, which couples variable names with instructions of type Alloc representing the address.

Translation of an assignment is simple, resolve the address of the variable in the symbol table and, to this address, save the value. On the other hand, loading a variable is a bit tricky. The resolution of the address is the same, but the problem comes with the decision of which register type should be used to store the loaded value. For this purpose, a load instruction must also have a type specified apart from the address. The type is obtained from the ‘Typechecker.’

```
// TinyC                         // IR
                                 function:
int main(){                      main:
                                   bb_0:
    double a;                        r0: int = AllocL 1 (size: 1)
    a = 15.0;                        r1: double = LoadImm 15
                                     Store [r0] r1: double
    return cast<int>(a);//!15        r3: double = Load [r0]
}                                    r4: int = Truncate r3: double
                                     Return r4: int
```

Listing 2.6: Example of translation of variables from TinyC language to IR.

Type-checking is necessary for two reasons. The first is to check that the input code is type-sound. So we can warn a user that a mistake is in the input code, and abort which is performed in semantic analysis. The second, more important here, is to correctly construct the IR.

When mentioning variables and types, the following common thing regarding memory and storage is an array. Arrays are translated by using AllocL in a local context; otherwise, AllocG with size parameter resolved from constant expression defining the size of that array. This Alloc instruction also represents an address of the first element in the array. The instruction 'ElemAddrIndex' is used when accessing the array's content. This instruction was created to handle all management of memory that has the need to offset an address.

It gets interesting when we get to the array's size and the length's resolution while compiling constant expressions defining length other than only numbers. For the resolution of the length of the array, it is necessary to implement a recursive function that counts the length according to the Instructions created by AST. The number of elements (length) of an array is multiplied by the size of the type that the array contains. As we said, for basic types, it is simple; just one memory cell is required. A list of instructions that preserves constant expression follows:

- LoadImm
- BinOp
- UnOp
- Extend
- Truncate

Some of them can be of type double because we allow conversion, so it is necessary to keep types; therefore, an analogous function resolving constant

```
// TinyC                 //IR
int main(){              function:
                         main:
                             bb_0:
  int size = 10;           ...
  double arr[10];          r3: int = LoadImm 10
                           r4: int = AllocL 1 x r3: int  (size: 10)
                           r5: int = AllocL 1 (size: 1)
                           Store [r5] r4: int
  double b = 178.5;        ...
                           r10: double = Load [r8]
  arr[0] = b;              r11: int = LoadImm 0
                           r12: int = LoadImm 1
                           r13: int = ElemAddrIndex r5: int + r11: int↩
     x r12: int
                           Store [r13] r10: double
}                          ...
```

Listing 2.7: Example of translation of an array from TinyC language to IR.

doubles is also required. Other Instructions are not necessary to implement, while the code below does not pass lexical analysis done by the parser.

For simplicity, constant expressions are those that are not loaded from memory.

The next part related to memory management is pointers. The problem was similar when we wanted to access a value in an array. A compiler must translate access to an array into pointer arithmetic to achieve the desired address. However, we use dedicated instruction, the 'ElemAddrIndex,' that counts the address of the element for us. When requesting a pointer, it is the same as for variables when we want to load them but omit the Load instruction. As mentioned before, while translating identifiers, only the address is required.

The last part concerning storage is about structures. Translation of structures has several parts. Definition, declaration, member assign, and member access. The definition does not produce any code afterward but is necessary for structure size, member size, and mapping of members. The sum of the sizes of all members equals the total size of the structure. Members can be of basic types, structures, or arrays as well.

The declaration is simple since no initialization nor constructor is supported; every new structure is uninitialized. The only way to write a value to a member is by member assignment.

Member assignment needs to specify *Alloc* instruction that represents the base address of the structure and offset counted from how many members were before the current member that is being accessed. The access to the member — its base pointer is resolved similarly as a variable is, but with the difference that the structure alone cannot be loaded; that statement is not supported in that case. Only an address of the structure is returned because it is used for the subsequent resolution for the member access or assign. An analogy with lvalue for variables is the same with members – returning of address and value. In Example 2.8, you can see how the structure copy is translated. There could be a solution instead of dedicated instruction to do it manually one by one with load and store for each member. However, in the t86 target, there is no instruction for manipulation with a memory of greater size than one cell. However, this is a concern of instruction selection. Therefore it is better to have it as separate instruction in IR and manage translation later with a tool that is concerned with available instruction in the target.

```
// TinyC         //IR
struct Pair{     main:
    int a;         bb_0:
    int b;           r0: int = AllocL 3 (size: 3)
};                   r1: int = AllocL 2 (size: 2)
struct               r2: int = LoadImm 72
DoublePair{          r3: int = LoadImm 0
    Pair c;          r4: int = ElemAddrOffset r1: int + r3: int
    int a;           Store [r4] r2: int
};                   r6: int = LoadImm 75
                     r7: int = LoadImm 1
int main(){          r8: int = ElemAddrOffset r1: int + r7: int
  DoublePair dp;     Store [r8] r6: int
  Pair p;            r10: int = LoadImm 0
  p.a = 72;          r11: int = ElemAddrOffset r0: int + r10: int
  p.b = 75;          StructAssign r11: int = r1: int ( of size: 2)
                     r13: int = LoadImm 0
  dp.c = p;          ...
  ...
}
```

Listing 2.8: Example of translation of a structure from TinyC language to IR.

Functions

To make functions work, we need a few things. We have covered the function as a part of IR and its Basic Blocks already, but the function needs to be called; therefore, we must introduce Call instructions. We need to work with arguments for that function as well, and lastly, a function should be able to return a result. For that purpose, there is a Return instruction. Return instruction has one more purpose: to end execution in that function and transfer control flow back to the caller function.

Call instructions are separated into two types by the difference between classic (static) call and function pointer call. In the first case, it is possible to know the starting point of the called function statically. In the second case, the information must be obtained from a register. Distinguishing them is easy; the first is identified with a Symbol registered in defined Functions in IRBuilder. The separation is good because the known start of the static call can be hard-coded into the call instruction directly and does not need to be loaded, which is an optimization step because the load instruction takes us space in memory, code size, and target machine time and also this static call instruction is convenient for analysis.

When resolving how to construct the Call instruction, there is a requirement to supply arguments for the function. It gets complicated when a structure is passed. That means from the TinyC view, we are passing a copy of the structure. Therefore we need to allocate and copy the structure. If we do not pass the copy, it acts the same as passing a pointer.

2.3 Optimizer

This Optimizer has implemented optimization techniques stored in passes, such as constant propagation, strength reduction, and function inlining. These passes can be scheduled after each other to perform the optimizations. The optimization topic is also related to the topic of analysis. A framework from the course NI-APR is implemented and it manages to run forward and backward analysis. A concrete analysis is implemented, which is the liveness analysis. The results from the liveness analysis are helpful for coloring register allocation described in the section discussed in the back-end.

2.3.1 Analyses

An analysis framework was created. The thesis also implements Liveness and Constant Propagation analysis and 'declaration' analysis using the framework. Framework supports forward and backward analyses and implements a function for searching a fixed point. The algorithm is described in the figure 2.2

For the analysis, a lattice need to be defined, and templates of Flat Lattice, Powerset Lattice, and Map Lattice are prepared for use. For the analysis, also a specialized map was implemented in '/tvlm/analysis/lattice/map_lattice' with the extended functionality of a default element for the map.

All analyses use control flow and data flow information stored as a graph constructed above the IR representation.

Liveness Analysis

The liveness analysis gives information about registers whose values will be used; therefore, having them in registers is necessary. The analysis gives this information at every instruction. When an analysis is implemented, the essential differences are in the used lattice and the transfer functions.

```
x = (⊥, ⊥, ..., ⊥)
W = {$v_1$, ..., $v_n$}

while (W ≠ ∅) {
  $v_i$ = removeNext(W)
  y = $t_{vi}(x_i)$
  for ($v_j$ ∈ dep($v_i$)) {
    z = $x_i$ ⊔ y
    if ($x_j$ ≠ z) {
        $x_j$ = z
        W += $v_j$
    }
}
return x
```

Figure 2.2: The propagation worklist algorithm used in the analysis template to reach a fixed point.

The transformation function needs to follow the backward analysis format of version 'may.' Therefore the joining of incoming analysis states is using least-upper-bound of the two as a result.

Constant propagation

Constant propagation analysis tracks at each instruction all memory locations. The result is updated whenever a constant is present at each point where a variable is stored. When storing a user-provided value, a *top* ($\top$) is issued, and whenever the memory location is uninitialized, the result is *bot* ($\bot$).

The results from constant propagation analysis are used in constant propagation optimization.

2.3.2 Optimizations

Optimizations are used to augment the IR representation so it does the execution 'better.' In this thesis, the main optimizations presented are function inlining, constant propagation, and strength reduction.

Strength Reduction

The task of Strength reduction is to substitute operations with different ones that are processed faster than the original ones and produce the same result. For example, when a number is multiplied by the power of two, it is often faster for the target machine to do it with a bit-shift operation instead of an arithmetic operation — multiplication.

The strength reduction needs to run an analysis that finds such instruction possible to replace. After marking the instructions, a substitution must be performed. Hence, all registers keep relations, and the original data flow is not harmed in a way that would cause the computation to be incorrect.

List of expressions that are substituted:

- multiplication of power of two with bit-shift
- division by power of two with bit-shift
- modulo by power of two with bit and

Constant Propagation

The constant propagation task is substituting instructions with the result already known in compile time so the computation is not impeding the final program. The task of the analysis part is to mark the instructions that need replacing and determine the results of such parts. When a variable read from memory is made, but the value is constant throughout, replacing the read with the actual value is possible, so slow access to the memory does not have to be invoked. The same applies when there is a binary operator with both operands constant; the constant result can be computed in compile time and replaced.

List of expressions that are substituted:

- All binary operators with constant operands
- All unary operators with constant operand
- Truncate instructions with constant operand with the integer type
- Extend instructions with constant operand with the floating-point type

Function Inlining

Function inlining has a task to eliminate the function call agenda and preserve the program's functionality. The function call agenda often causes extra work for the target machine, especially in cases when the function has only a few instructions; therefore, it is suitable to skip the time-consuming part and only execute the body. The reason why programmers use function calls instead of having the function body in code is that copied code cannot be changed in one place; therefore, the function inlining is a valuable tool to have, while programmers can have sustainable code and do not pay for it in terms of code efficiency.

Function inlining is divided into 5 phases: prolog, phase1sthalf, inlining, phase2ndhalf, and epilogue. Each phase defines what needs to be done with Basic Block in the function that contains a call to the inlined function.

The function, which has the call of the inlined function, is divided into three parts (batches) by Basic Blocks, the first part contains all Basic Blocks preceding the one Basic Block containing the call to inlined function, and the third contains the rest of Basic Blocks. All jumps need to be transformed so they point to new Basic Blocks.

The *prolog* phase prepares new Basic Blocks for the function that contains the call of the inlined function and fills all Basic Blocks from the first batch with copied contents of old Basic Blocks.

The *phase1sthalf* phase copies the first half of the Basic Block in the second batch and inserts a jump to the first Basic Block that the *inlining* process fills.

The *inlining* phase copies all Basic Blocks of the inlined function. It replaces the return instruction with an adequate jump instruction to the next Basic Block and stores the result.

The *phase1sthalf* phase copies the second half (after the replaced call) of the Basic Block in the second batch. It also replaces the instruction that uses the result of the inlined call instruction with an instruction that uses the stored result instead.

The *epilog* copies the contents of all Basic Blocks in the third batch.

2.4 Back-end

The back-end is the last part of this compiler. The back-end input can be code generated by the front-end or optimized from the middle-end; In any case, the program is in the IR. The task of the back-end is to transform this IR into Assembler.

The task here is to create a result represented as an Assembly code. The compiler must select assembler instructions and choose registers used in the target accordingly. Moreover, in the last phase, it should clear suboptimal parts with Peepholer; its place is after the instruction selection or after the register allocation is finished. While compiling the whole program, it is not sufficient just to translate (and select) instructions, Basic Blocks, and functions. Keep in mind, in the back-end; there are more problems resolved: concerning calling conventions, information about registers, working with addresses and memory of the target machine, as well as addresses of instructions for jumps because assembler jump instructions jump within the Assembler code, which does not contain Basic Blocks but labels.

The process of target code generation is divided into parts. Chronologically listed in this order:

Instruction selection (IS) where adequate assembler instructions are selected. IS has the most significant influence on the final code structure.

Peepholer is responsible for clearing and repairing some suboptimality resulting from instruction selection.

Register Allocation (RA) must operate after the instruction selection is finished. It requires the number of how many and which registers the target machine has. From the selected program, it gets information about used IR registers and tries to fit them in the registers. In case of a lack of target registers, it has to change the program so it is possible to fill them with the IR registers, and the program stays correct.

Assembly finalization. This part takes all results from all preceding sections and creates a complete assembler program suitable for the target machine.

2.4.1 Instruction Selection

The first part of the back-end is instruction selection (IS). IS aims to choose how to translate every IR instruction in the input to the target instruction or a sequence of them. Which target instructions are available specifies the target instruction set. The decision of which target instruction to pick often depends on constraints like the efficiency or code size of the outputted program. These constraints naturally go against each other, making this quite a challenging task.

This work implemented naive IS managed by visitor-pattern on IR representation — each instruction creates a sequence of assembler instructions which are then (in Assembly finalization) composed together after each other following the order in each Basic Block. Each Basic Block then follows how they are ordered in function. And functions by the same pattern as they are ordered in the program. The program has to make a selection for global Basic Block as well.

The IS selects instructions for each Basic Block. Selection is made by order of the sequence for each instruction. IR instruction is generally translated into one or more target instructions. Some IR instructions are only virtual because IS does not select any target instruction, mainly because the role of such IR instruction is to signal additional information to the code generation process. Basic Block can be safely independently translated, and the translation will be correct while compiled in the same order as they are ordered in the Basic Block. This order is correct since instructions cannot have arguments later in the sequence. The independent selection is not necessary, nor reasonable for the virtual instructions.

For example, *Load* and *Store* instructions take into account the *Alloc* instruction, which belongs to the Virtual ones, to distinguish how the address should be obtained. And still, the independent view has a downside in terms of optimization while translating instructions as dependent; a better sequence of target instructions might be constructed, but generally, in the trade of demanding design, that can be pretty time-consuming when evaluated, causing greater compile times.

```
// IR                                        // Target Code
                                              0: JMP 11
main:                                         1: PUSH Bp
  bb_0:                                       2: MOV Bp, Sp
                                              3: SUB Sp, 1
    r0: int = LoadImm 5                       4: MOV Reg1, 5
    Store [g0] r0: int                        5: MOV [0], Reg1
    r2: int = Load [g0]                       6: MOV Reg1, [0]
    Return r2: int                            7: MOV Reg0, Reg1
                                              8: MOV Sp, Bp
                                              9: POP Bp
globals:                                     10: RET
  g0: int = LoadImm 17                       11: MOV Reg1, 17
  g1: int = AllocG 1 (size: 1)               12: MOV [0], Reg1
  Store [g1] g0: int                         13: MOV Reg0, 1
                                             14: CALL 1
                                             15: HALT
```

Listing 2.9: Example of translation of an AllocG instruction from IR to Tinyx86.

On the other hand, the independent view is simple to implement, and debugging is also simpler. Moreover, another advantage is that the process is more understandable for students.

To understand future examples, it is advised to see the Instructions section in the Tiny x86 ISA chapter of this thesis (named Tiny x86 - Architecture Simulator for Educational Purposes)[11] with the specification of ISA of the t86 target.

The first Example 2.9 is how are *AllocG* instructions translated. Both of the *Alloc* instructions are purely symbolic, they do not contribute to the Assembler code as other IR instructions by creating target instructions. The *Alloc* instructions affect only counters inside the compiler that need to know how much of allocated space there needs to be created and to map, which pairs the *Alloc* instructions with an offset at which the allocated place is. Therefore whenever a *Load*, *Store*, or *ElemAccess* instructions work with address a routine is called which in case of *Alloc* instruction counts and gathers the address accordingly. The address is a relative offset from the Base pointer, in the case of AllocL, and just an offset from the bottom of the memory in the case of AllocG.

AllocL is tightly connected to functions. Alloc instructions are virtual; they provide information for the Assembly finalization, so then the finalization part can specify local memory needs for the function and insert instructions responsible for the memory allocation.

The IS has a routine that helps translate instructions that work with the address. The routine distinguishes the types of allocation, and in case it is Alloc instruction, it fills it in place without additional registers.

Translation of a function requires a compilation of all Basic Blocks in any order. However, the compilation order is deterministic since Basic Blocks are stored in a vector. It works by the sequence in which they are stored. The function compilation needs to count with memory, particularly stack. Since the '*CALL*' and the '*RETURN*' target instruction work with the stack and local allocations. At the start and the end, as well as in every function, there needs to be a code sequence that defines stack preparation and movement. This sequence can be seen in the Example 2.10. The sequence is there to set up the stack place so that it can be reverted after the call body is executed and resume correctly in the caller function. That is, the standard usage of Stack Pointer (Sp) and Base Pointer (Bp) registers for managing the stack workaround. (Instructions such as `PUSH`, `POP`, `CALL`, `RETURN`, are working with Sp and Bp)

The stack movement has a few common patterns used in specific use cases:

- at the begging of a function,
- at the end of a function,
- preparation of arguments for a function call,
- accessing function arguments and local variables,
- and local variable address management.

The stack is the structure for storing local memory; therefore, the stack is the middle piece between the function that prepares the stack and AllocL instruction which points to places where the variables are or will be stored.

Even more interesting is how *AllocL* has to be translated. Every start of a function has a mandatory sequence for working with a stack. The sequence prepares special registers for managing local allocation placed on the stack. If it is not resolved this way and only allocating whenever the instruction is hit, it would cause unnecessary allocations and waste of stack space. The potential combination with other IR Instructions, which form loops or others that change the execution flow of the program, is problematic. Imagine when a variable is allocated inside a loop. That would mean it is allocated that many times there were cycles in a loop, effectively running out of stack space relatively quickly. It would not make any sense because the value should still use the same address, and there would be no way to address the memory from the loop allocated instances.[4] Therefore when compiling, Alloc instructions change counters of how many allocations there are in the function. This value is used when assembling the function and adding allocation info (which happens in Assembly finalization). This way, the issue with loops is solved, while everything is allocated only once and at the beginning of the function as desired.

[4]There is only one way — using pointer arithmetic's, which is forbidden to use in tinyC anyways.

```
// IR                                          // Target Code
                                               0: JMP 30
main:                                          1: PUSH Bp
                                               2: MOV Bp, Sp
  bb_0:                                        3: SUB Sp, 4
   r0: int = AllocL 1 (size: 1)
   r1: int = LoadImm 0                         4: MOV Reg1, 0
   r2: int = AllocL 1 (size: 1)
   Store [r2] r1: int                          5: MOV [Bp + -2], Reg1
   Jump bb_1                                   6: JMP 7
  bb_1:
    r5: int = Load [r2]                        7: MOV Reg1, [Bp + -2]
    r6: int = LoadImm 5                        8: MOV Reg2, 5
    r7: int = Lt r5: int r6: int               9: SUB Reg1, Reg2
                                              10: MOV Reg1, Flags
                                              11: AND Reg1, 1
    CondJump r7: int bb_3 bb_2                12: CMP Reg1, 0
                                              13: JZ 20
                                              14: JMP 15
  bb_2:
    r9: int = LoadImm 5                       15: MOV Reg1, 5
    r10: int = AllocL 1 (size: 1)
    Store [r10] r9: int                       16: MOV [Bp + -3], Reg1
    r12: int = Load [r10]                     17: MOV Reg1, [Bp + -3]
    Store [r0] r12: int                       18: MOV [Bp + -1], Reg1
    Jump bb_4                                 19: JMP 25
  bb_3:
    r20: int = Load [r0]                      20: MOV Reg1, [Bp + -1]
    Return r20: int                           21: MOV Reg0, Reg1
                                              22: MOV Sp, Bp
                                              23: POP Bp
                                              24: RET

  bb_4:
    r15: int = Load [r2]                      25: MOV Reg1, [Bp + -2]
    r16: int = Copy r15: int                  26: MOV Reg0, Reg1
    r17: int = Inc  r15: int                  27: INC Reg1
    Store [r2] r17: int                       28: MOV [Bp + -2], Reg1
    Jump bb_1                                 29: JMP 7

                                              30: MOV Reg0, 1
                                              31: CALL 1
                                              32: HALT
```

Listing 2.10: Example of translation from IR to Tinyx86, where an AllocL instruction is in a loop.

Translating calls concerns mainly work with the call's arguments. The register allocation tasks include calling conventions and spilling the suitable registers. instruction selection only prepares arguments to the proper registers or pushes them to the memory. So *ArgAddr* in callee points to the correct memory place.

When it comes to *Jump*, *CondJump*, or *Call* instructions, there is one thing they have in common, their argument is the place where the execution of the program has to continue, the destination. In assembler, that is solved by supplying a number — the position of the destination instruction in the program. Unfortunately, this information is not available at the time of instruction selection is made — the magical number 42 in assembler instruction at position 59 in the Example 2.11 shows this, therefore a mapping needs to be created between the jump (or call) instruction and Basic Block (or Function). This mapping is then processed in the Assembly finalization when all instructions are selected, and the positions are known.

2.4.2 Register Allocation

Imagine the register allocator as an interface providing a function to get a register for a specific instruction; here comes in handy that those instructions are the same as the values (results) that need to be placed in a register. When instruction selection is finished, placeholders are mapped the most straightforward way whenever a register is required to get the next one with an index higher. That part is named a 'Register Assigner.' The task of the register allocator is to remap these register placeholder numbers with some from the set corresponding to a number of registers available in the target machine.

Each instruction behaves differently but uses the same interface. If a register is required, ask for remapping a particular placeholder by calling the function 'setupRegister.' That function looks into the Register Allocator's structures and decides which register number the placeholder will use from now on. The result of this function depends on which algorithm is used.

When it comes to Register Spilling, a Register Allocator needs to maintain values and work with them. It involves a few types of demands

- Is the value already in some register? And which one?
- Is there a free register? And which one?
- Modify the program to restore a value into a specified register.
- Decide which register needs to be spilled and spill it.

Register spilling occurs when an instruction needs a free register to store its result, but every register is already filled. Therefore one result needs to be swapped out to memory. Therefore it is required that RA can modify the program.

```
// IR                                     // Target Code
function:                                  0: JMP 57
fact:                                      1: PUSH Bp
                                           2: MOV Bp, Sp
    bb_0:                                  3: SUB Sp, 3
        r0: int = ArgAddr 0                4: MOV Reg1, Bp
                                           5: ADD Reg1, 2
        r1: int = Load [r0]                6: MOV Reg2, [Reg1]
        r2: int = LoadImm 0                7: MOV Reg3, 0
        r3: int = Eq r1: int r2: int       8: SUB Reg2, Reg3
                                           9: MOV Reg2, Flags
                                          10: AND Reg2, 2
        CondJump r3: int bb_2 bb_1        11: CMP Reg2, 0
                                          12: JZ 19
                                          13: JMP 14
  bb_1: r5: int = LoadImm 1               14: MOV Reg2, 1
        Return r5: int                    15: MOV Reg0, Reg2
                                          16: MOV Sp, Bp | 17: POP Bp
                                          18: RET
  bb_2: r8: int = Load [r0]               19: MOV Reg2, [Reg1]
        r9: int = Load [r0]               20: MOV Reg3, [Reg1]
        r10: int = LoadImm 1              21: MOV Reg1, 1
        r11: int = Sub r9: int r10: int   22: SUB Reg3, Reg1
        r12: int = CallStatic             23: MOV Reg0, Bp
                    fact (r11: int)       24: SUB Reg0, 1
                                          25: MOV [Reg0], Reg3
                                          26: MOV Reg0, Bp
                                          27: SUB Reg0, 2
                                          28: MOV [Reg0], Reg2
                                          29: PUSH Reg3
                                          30: CALL 1
                                          31: MOV Reg1, Reg0
                                          32: ADD Sp, 1
                                          33: MOV Reg2, Bp
                                          34: SUB Reg2, 2
                                          35: MOV Reg2, [Reg2]
        r13: int = Mul r8: int r12: int   36: MUL Reg2, Reg1
        Return r13: int                   37: MOV Reg0, Reg2
                                          38: MOV Sp, Bp | 39: POP Bp
                                          40: RET
 bb_3:  Jump bb_2                         41: JMP 19
main:                                     42: PUSH Bp
                                          43: MOV Bp, Sp
                                          44: SUB Sp, 2
  bb_0: r15: int = LoadImm 3              45: MOV Reg1, 3
        r16: int = CallStatic             46: MOV Reg0, Bp
                    fact (r15: int)       47: SUB Reg0, 1
                                          48: MOV [Reg0], Reg1
                                          49: PUSH Reg1
                                          50: CALL 1
                                          51: MOV Reg2, Reg0
                                          52: ADD Sp, 1
        Return r16: int                   53: MOV Reg0, Reg2
                                          54: MOV Sp, Bp | 55: POP Bp
                                          56: RET
                                          57: MOV Reg0, 1
                                          58: MOV Reg0, 42
                                          59: CALL 42
                                          60: HALT
```

Listing 2.11: Example of translation of IR with Call expressions to Tinyx86.

In this work, there are two designs of register allocator

Naive Allocator – register allocator that uses a queue for deciding which register to spill. It tries to divide the occupancy between all registers evenly. It is considered naive because it does not care about the usage of the registers.

Coloring Allocator – register allocator that builds an internal representation of dependencies and uses an analysis to determine which register is the best candidate for spilling, thus optimizing against the number of spilled registers.

The Naive register allocator spills the last defined values, so it maintains time locality (kind of) but is easy to implement.

On the other hand, the Coloring allocator uses a more complicated structure and liveness analysis to determine which values are best to spill according to colliding requirements for being in registers. The coloring allocator inherits spilling mechanisms implemented in the Naive register allocator.

The workflow of the Coloring allocator is following:

1. Liveness analysis is performed. It collects information about unique live ranges.
2. Assign Virtual registers information about the live ranges
3. Compute — Color the live ranges — and determine which live ranges will be spilled; if any, insert spill code and redo from the start with the analysis.
4. Assign the virtual registers Physical registers

2.4.3 Calling Conventions

Calling conventions are rules about calling a function and returning from a function. It defines an interface that the caller and the callee need to obey. If any would not, the computation can be harmed. The rules are about

They define

- how are the arguments passed into the function,

  The arguments are all pushed to a stack, so every argument has a place in memory and *ArgAddr* instruction points to the memory place.

- who cares about which registers and who is responsible for saving them in case of using and rewriting the values.

  The caller saves everything, and the callee has a free space to work with.

- how the workflow around the stack works.

  At the beginning of every function, there is the Base pointer marked — Pushed Bp, then into Bp saved the Bp. After compiling the *Return* instruction complement sequence for resetting the stack and, therefore, 'deallocating' all locally allocated values in that function. The sequence is to replace Sp with Bp, POP to Bp, and then call RET to exit from the callee and resume in the caller.

2.4.4 Assembly Finalization

The *Assembly finalization* is the last part of code generation. Its task is to take all parts of the results from the preceding sections and path them together.

Global instructions (globals) are in separate Basic Block in the program. IS compiled them before as their contents are necessary for addressing global variables. However, Assembly finalization needs the globals at the end because it has to fill in information about functions and where they begin in the assembler code. Therefore compilation of globals is postponed and takes place at the end of the assembler code, where the control flow is patched by jumps.

The Assembly finalization begins with composing the program prologue routine of jumping to globals, setting the context for the main function.

After that, function composition is the next step. Functions already have the selected code tied to their components. Assembler sequences created in IS are put after each other following the order in each Basic Block. Each Basic Block then follows the order of how they are ordered in function. And functions, by the same pattern, whole chunks are after each other, the way they are ordered in the program.

Compiling a Basic Block has one catch, the label of the first instruction has to be remembered, in case the *Jump* instruction is targeting that Basic Block or if it is the first Basic Block of a function, then it is required for *Call* translation.

After the parts mentioned above are done, the main context and all functions are composed and globals afterward. The last thing remains to be finished, and that is Jump and Call instructions patching. Throughout the IS, whenever a jump or call should be selected in destination, there was a filled empty label waiting until now. Now the information from function composition remembered where each Basic Block begins in the composed code; this information can be substituted, creating a finished target machine code.

Chapter 3

Evaluation

This chapter shows the differences between the naïve compilation and compilation with the following optimization techniques: function inlining, constant propagation, strength reduction, dead code elimination, and a register allocator that uses graph coloring for spill decision. The benefits of performance gained are showcased in section 3.2.

3.1 Goals and Setup

The goal was to create an optimizing compiler that implements techniques from the NI-GEN course. Furthermore, they can be used for illustration of the implemented techniques. The whole program is a composition of the compiler and the target VM. It executes the compiler result right after the compilation is successfully finished. The composite contains: loading an input file to the compiler, compilation itself, and target VM execution of the compiler result.

The tests are evaluated on tiny86 Virtual architecture, which provides performance counters. The performance counters deliver information about the execution in the target virtual processor ticks.

3.2 Verification and Validation

The compiler was tested with included tests in the tinyC repository. All tests were successfully translated into the tiny86 instructions (with the default arguments). Testing scripts are also included in the */tinyc/tests/* directory.

Sample A		
	Optimized run	Naïve run
Total ticks:	202	684
Total code instructions:	76	174
Throughput: [ins. per tick]	0.376238	0.254386

Table 3.1: Benchmark comparison of code sample A, where there are results of the optimized compilation on the left, and on the right, there are results from the most Naïve run the compiler can perform. All units are ticks unless specified otherwise.

The program has several arguments to change the settings which determine the enabled optimization. Changeable arguments that have an impact on the optimizations and the outputted program are:

Arguments for Target

- The number of int and float registers. The register allocator's results depend on target register counts. That changes the register pressure and affects how often registers need to be spilled.

Arguments for compiler

- Swap between register allocations - color and naive
- Turns on and off the optimizations
  - Constant propagation (& strength reduction)
  - Function inlining
  - Dead code elimination

Testing in loops is unnecessary, thanks to the tiny86 Virtual Machine, which provides consistent results while measured in ticks that are not dependent on time or other constraints defining the targets speed.

For the showcase of the results, two representative code samples were chosen; they are shown in Example 3.1. Results of the performance counter are shown in tables 3.1 and 3.2. Turning the optimizations on lowers the ticks necessary to achieve the same result of the execution. Improvement is in total executed instructions The amount of ticks required to finish the computation is also lower. In sample A, the necessary ticks are one-third of the amount that the naïve compilation's result required. For sample B, the optimized program required half of the instruction to produce the same result.

Code sizes also improved; the naïve compilation required 132 instructions, whereas the optimized required 76. For sample B, 116 instructions were outputted from naïve compilation, and only 72 with optimizations enabled.

```
//Code sample A
int add(int a, int b ){
  return a + b;
}

int sub(int a, int b){
  return a - b;
}

void prnt(int a ){
  char b = cast<char>(a);
  print(b);
  print('\n');
  return;
}

int main(){
  int a = 85;
  int b = 10;

  prnt(add(a, b));
  prnt(sub(a, b));
  prnt(a + b);
  prnt(a - b);
  return a+b;
}
```

```
// Code sample B
int abs(int n) {
  if (n < 0) {
    return -n;
  } else {
    return n;
  }
  return 0;
 }

 int main() {
   int n = 4+1;
   int m = -7;

   int abs_n = abs(n);
   int abs_m = abs(m);

   int k = 10;
   int n_times_k = n * k;
   int m_times_k = m * k;

   return m_times_k;
 }
```

Listing 3.1: Code samples used for testing, sample A (on the left) was used for constant propagation, and sample B (on the right) was used for function inlining.

Sample B		
	Optimized run	Naïve run
Total ticks:	302	521
Total code instructions:	79	138
Throughput: [ins. per tick]	0.261589	0.264875

Table 3.2: Benchmark comparison of code sample B, where there are results of the optimized compilation on the left, and on the right, there are results from the most Naïve run the compiler can perform. All units are ticks unless specified otherwise.

Creating optimal testing microbenchmarks is a difficult task. Tested programs were also used for testing the coverage of the tinyC translation. In a larger program, there could be a more significant possibility to optimize more sections. So the results can differ depending on the tests and what functionality is targeted, i.e., programs do not have functions; therefore, a function inlining will not make any difference.

Nevertheless, the testing was performed. The average code size among all tested programs (including these two representative code samples) decreased by 30%, and the average amount of ticks necessary to execute decreased by 31%. Graphs 3.1, 3.2 show a detailed overview of how the tests performed and the impact of optimization on performance. The worst case of the measured cases is that the optimization does not change the code; therefore, the graphs do not contain 'negative' results. The two types of RA make a significant difference in *13spilling* while under big register pressure.

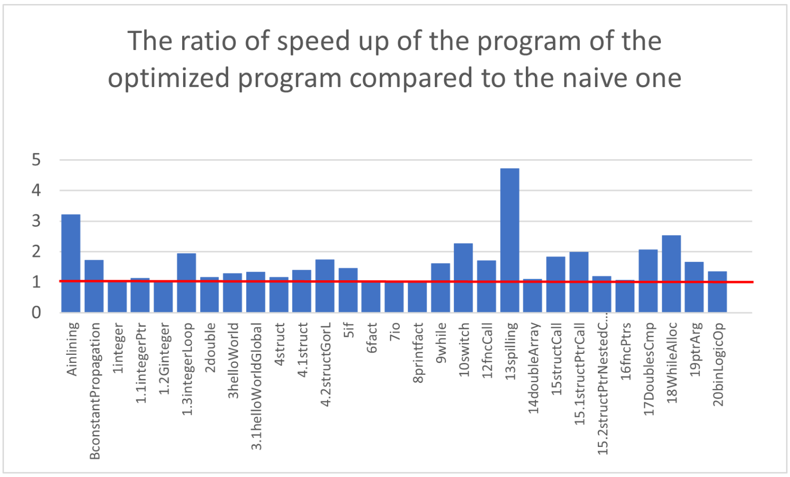


Figure 3.1: Graph showing the ratio of speed up of the program of the optimized program compared to the naive one.

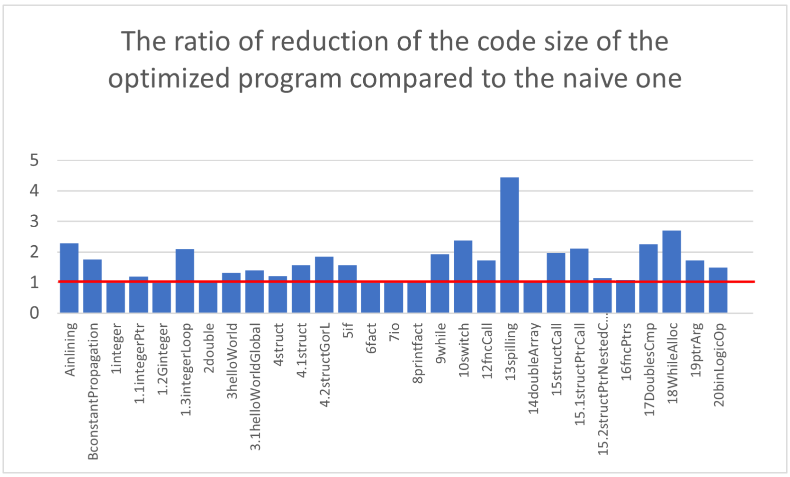


Figure 3.2: Graph showing the ratio of reduction of the code size of the optimized program compared to the naive one.

Conclusion

The optimizing compiler for the tinyC language and tiny86 target was designed and implemented. The implementation uses techniques from the NI-GEN course. In the middle-end part, intra-procedural optimizations were implemented, including dead code elimination, constant propagation, and function inlining. It implements instruction selection and two types of register allocation in the back-end part. The compiler supports configuration to toggle various optimization techniques described in this thesis, so the comparison between naïve and optimizing compilation can be measured. The implementation was tested by unit tests included in the implementation.

Future Work

The middle-end can be upgraded in many ways beginning with extra analysis and optimization techniques that would optimize the code even more. In that case, a scheduler could use an upgrade as well. The compiler is prepared for further implementation of Peepholer in the back-end. Also, another algorithm for instruction selection can be implemented so a comparison between the naïve and the new algorithm can be measured. The compiler parts are modular so that another implementation of any part of the compiler can be upgraded since the underlying tiny86 VM allows one to compare two implementations easily. Finally, a compiler, although a small one as presented in this thesis, is complex software, and while care was taken to test its features properly, those tests are by no means complete. Thesis coverage should be increased, and any bugs found fixed.

Bibliography

[1] Aho, A. V.; Sethi, R.; et al. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Rice, H. G. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, volume 74, no. 2, 1953: pp. 358–366, ISSN 00029947. Available from: `http://www.jstor.org/stable/1990888`

[3] Chaitin, G. J.; Auslander, M. A.; et al. Register allocation via coloring. *Computer Languages*, volume 6, no. 1, 1981: pp. 47–57, ISSN 0096-0551, doi:https://doi.org/10.1016/0096-0551(81)90048-5. Available from: `https://www.sciencedirect.com/science/article/pii/0096055181900485`

[4] A brief history of GCC. [Online; accessed 12-October-2022]. Available from: `https://gcc.gnu.org/wiki/History`

[5] Tree SSA (GNU Compiler Collection (GCC) internals). [Online; accessed 10-October-2022]. Available from: `https://gcc.gnu.org/onlinedocs/gccint/Tree-SSA.html`

[6] The LLVM Compiler Infrastructure Project. [Online; accessed 17-October-2022]. Available from: `https://llvm.org`

[7] Getting started with the LLVM system. [Online; accessed 17-October-2022]. Available from: `https://llvm.org/docs/GettingStarted.html`

[8] Writing an LLVM backend. [Online; accessed 19-October-2022]. Available from: `https://llvm.org/docs/GettingStarted.html`

[9] LLVM language reference manual. [Online; accessed 5-January-2023]. Available from: `https://llvm.org/docs/LangRef.html`

[10] Wikipedia. Short-circuit evaluation — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Short-circuit%20evaluation&oldid=1130491239`, 2023, [Online; accessed 20-February-2023].

[11] Strejc, I. *Tiny x86 - Architecture Simulator for Educational Purposes*. Master's thesis, Czech Technical University in Prague, 2021.

Appendix A

Acronyms

ASCII	American Standard Code for Information Interchange
AST	Abstract syntax tree
BB	Basic Block
DAG	Directed acyclic graph
IR	Intermediate Representation
IS	Instruction Selection
ISA	Instruction set architecture
SSA	Static single assignment

Appendix **B**

TinyC Language Specification

B.1 TinyC Language Reference

```
PROGRAM := { FUN_DECL | VAR_DECLS ';' | STRUCT_DECL |
           FUNPTR_DECL }
```

A program is collection of function and variable declarations. Alternatively, *tinyC* supports a repl-like interface, where the main element is either a program, or a stetement:

```
REPL := STATEMENT | PROGRAM
```

B.1.1 Functions

```
FUN_DECL := TYPE_FUN_RET identifier '(' [ FUN_ARG
            { ',' FUN_ARG } ] ')' [ BLOCK_STMT ]
FUN_ARG := TYPE identifier
```

Functions are declared the same way as in C. A function definition starts with the return type, followed by function name and arguments. Arguments have type and name. Default values or variadic arguments are not supported. Functions can be declared or defined. Function definitions must have their body.

B.1.2 Statements

```
STATEMENT := BLOCK_STMT | IF_STMT | SWITCH_STMT |
             WHILE_STMT | DO_WHILE_STMT | FOR_STMT |
             BREAK_STMT | CONTINUE_STMT | RETURN_STMT |
             EXPR_STMT

BLOCK_STMT := '{' { STATEMENT } '}'
```

```
IF_STMT := if '(' EXPR ')' STATEMENT [ else STATEMENT ]

SWITCH_STMT := switch '(' EXPR ')' '{' { CASE_STMT }
            [ default ':' CASE_BODY } ] { CASE_STMT } '}'
CASE_STMT := case integer_literal ':' CASE_BODY
CASE_BODY := { STATEMENT }

WHILE_STMT := while '(' EXPR ')' STATEMENT
DO_WHILE_STMT := do STATEMENT while '(' EXPR ')' ';'
FOR_STMT := for '('[EXPR_OR_VAR_DECL]';'[EXPR]';'[EXPR]
            ')' STATEMENT

BREAK_STMT := break ';'
CONTINUE_STMT := continue ';'

RETURN_STMT := return [ EXPR ] ';'

EXPR_STMT := EXPR_OR_VAR_DECL ';'
```

B.1.3 Types

To keep the type declarations similar to those of C while keeping the grammar simple to parse, the type declarations grammar is a bit repetitive:

```
TYPE := (int | double | char | identifier) { * }
     |= void * { * }

TYPE_FUN_RET := TYPE | void
```

Pointers can point to either a plain type, an identifier representing a struct or function pointer type, or `void`. Pointers to pointers are allowed.

B.1.4 Type Declarations

```
STRUCT_DECL := struct identifier ['{'{ TYPE identifier
               ';'}'}']';'
```

Structured types must always be declared before they are used. Forward declarations are supported as well.

```
FUNPTR_DECL := typedef TYPE_FUN_RET '(' '*' identifier
               ')' '(' [ TYPE { ',' TYPE } ] ')' ';'
```

Function pointers must always be declared before they can be used.

B.1.5 Expressions

```
F := integer | double | char | string | identifier |
     '(' EXPR ')' | E_CAST
E_CAST := cast '<' TYPE '>' '(' EXPR ')'

E_CALL_INDEX_MEMBER_POST := F { E_CALL | E_INDEX |
                               E_MEMBER | E_POST }
E_CALL := '(' [ EXPR { ',' EXPR } ] ')'
E_INDEX := '[' EXPR ']'
E_MEMBER := ('.' | '->') identifier
E_POST := '++' | '--'

E_UNARY_PRE := { '+' | '-' | '!' | '~' | '++' | '--' |
               '*' | '&' } E_CALL_INDEX_MEMBER_POST
```

Unary prefix operators are either simple arithmetics (plus, minus, not and neg), the increment and decrement operators (these are a bit harder to compile as they immediately update the value of their argument) and the special operators for dereferencing ('*') and getting an address of variables ('&').

```
E1 := E_UNARY_PRE { ('*' | '/' | '%' ) E_UNARY_PRE }
E2 := E1 { ('+' | '-') E1 }
E3 := E2 { ('<<' | '>>') E2 }
E4 := E3 { ('<' | '<=' | '>' | '>=') E3 }
E5 := E4 { ('==' | '!=') E4 }
E6 := E5 { '&' E5 }
E7 := E6 { '|' E6 }
E8 := E7 { '&&' E7 }
E9 := E8 { '||' E8 }
```

Basic arithmetic operators are supported in the same way they are in `c/c++` including their priority. All evaluate left to right (left associative).

```
EXPR := E9 [ '=' EXPR ]
EXPRS := EXPR { ',' EXPR }
```

The lowest priority is the assignment operator. Note that assignment operator is right-to-left (right associative).

Expressions can be separated by a comma, they will be evaluated left to right.

```
VAR_DECL := TYPE identifier [ '[' E9 ']' ] [ '=' EXPR ]
VAR_DECLS := VAR_DECL { ',' VAR_DECL }
```

```
EXPR_OR_VAR_DECL := VAR_DECLS | EXPRS
```

Variable declaration must start with a type specification. Multiple variables of same type cannot be declared in a single expression, but multiple comma separated declarations with explicit type are allowed.

Optionally, arrays of statically known size may be defined with `[]` operator after the variable name.

Appendix C

IR Instructions

This chapter follows with a description of supported instructions in Intermediate Language, which is split into sections by their purpose.

C.1 Control Flow Instructions

Control Flow Instructions are those that change the order of instruction's execution. Those can be divided into Function calls and terminator Instructions. Terminator instructions are those which are always last in Basic Block. Then we can call such a Basic Block closed.

Return

```
Return 'value' : 'type' //Return a value from a non-void ↩
    function
Return  //Return from a void function
```

'Return' is Terminator instruction and serves as the ending of a function. When the 'Return' instruction is executed, the control flow returns back to the calling functions context. If the instruction returns a value, that value, which is specified in the argument, shall set the 'Call' or 'CallStatic' instruction's return value.

Jump

```
Jump `targetBB'
```

Jump is Terminator instruction and changes the flow of execution to another Basic Block in the current Function. It corresponds to an unconditional branch.

CondJump

```
CondJump 'cond': 'type' 'falseBB' 'trueBB'
```

CondJump instruction is similar to *Jump* instruction as well as a Terminator instruction but the difference is that this instruction takes two Basic Blocks (a true and false one) and a condition by which is then decided to which of that two should the execution continue. So if the condition equals 0 then false Basic Block will be next to execute, and otherwise, true Basic Block is used for the next execution. It corresponds to a conditional branch. After the execution of a conditional 'CondJump' instruction, the 'cond' argument is evaluated. If the value is true, control flows to the 'trueBB' Basic Block argument. If 'cond' evaluates false, control flows to the 'falseBB' Basic Block argument.

Halt

```
Halt
```

Halt is a simple Terminator instruction that serves just one purpose: To stop the machine and end its execution, otherwise it would do 'NOP' instruction in an endless loop. So this instruction is always the last at the end of the program. Stops the execution of the program.

C.2 Function Call Instructions

Since tinyC supports function pointers, there need to be two types of instructions for calling a procedure. The first is named CallStatic because its address should be known by the compiler at compile time. And second, named simply Call for calling a procedure that is not known at compile time. A great analogy to this problem is with storing a value in memory, but the value now is the address of the first instruction in that procedure.

CallStatic

```
r1:'rType'=CallStatic 'fnc'(['function args')
CallStatic 'name'(['function args')
```

CallStatic instruction represents a function call. It takes a Function and a collection of values that will be used as an argument. This instruction is used to cause control flow to transfer to a specified function, with its incoming arguments bound to the specified values. Upon a 'Return' instruction in the called function, control flow continues with the instruction after the function call, and the value of this instruction becomes the return value of the function.

Call

```
r1:'rType'=Call 'value'('function args')
Call 'value'('function args')
```

‘Call’ instruction is similar to the CallStatic instruction but differs in the information that is known about the function that will be called. Instead of Function, it takes a Instruction Label that should point to a start of a function, in other words, the first instruction in the function that the called function pointer points to.

C.3 Memory Access & Allocation Instructions

For purposes of memory allocation, TvlmIR supports basic types. They are used for the calculation of how much memory needs to be allocated to store the type. And for Structure representation so accesses to members can be calculated.

Note that the target machine that we are compiling to, there exist only two types of registers that hold a full-fledged value of type integer or float, so These types will be reduced in a way that they correspond with the source, but not precisely. Also, they take up always one cell of memory, so there isn’t any hassle with storing different lengths of values. Only Structs and Arrays can be of different lengths.

Tvlm Types in IR are:

Integer normally occupies 4 bytes of memory.

Double normally occupies 8 bytes of memory. In target, we can only work with float arithmetic, therefore we substitute double precision with single precision floats.

Char occupies 1 byte of memory.

Array construction needs a base Type, and ‘index’; an instruction that evaluates into an integer that will represent the size of this array. Size is computed as follows: ‘base type size’ times ‘index’.

Pointer construction needs a base type. The size is equal to 4 bytes.

Struct construction needs a specification of pairs ‘Symbol’ with ‘Type’. Every pair represents a field inside the defined structure. Size is computed as a sum of all fields added together. Note: empty ‘Struct’ is correct and its size is 1 because every object needs to have an address. Structs can’t have a field of type Array.

Void is placeholder type without size. It is necessary for a pointer, so there is a possibility to instantiate ‘void *’ expression.

Tvlm Instruction follows:

AllocG

```
r2: int= AllocG 'size'('type') x 'numOfElems': int
r1: int= AllocG 'size'('type')
```

The *AllocG* instruction allocates memory in Data Segment at the global scope which means it can occur only in the global Basic Block. After execution memory is allocated (which happens at the beginning, before the main function call), then a pointer is returned. The allocated memory is uninitialized, and loading from uninitialized memory produces an undefined value. The operation itself is undefined if there is insufficient space. Allocating zero bytes is legal, but the returned pointer may not be unique.

AllocL

```
r2: int= AllocL 'size'('type') x 'numOfElems': int
r1: int= AllocL 'size'('type')
```

The *AllocL* instruction allocates memory on the stack frame of the currently executing function, to be automatically released when this function returns to its caller. After execution memory is allocated (which happens at the beginning, before the main function call), then a pointer is returned. The allocated memory is uninitialized, and loading from uninitialized memory produces an undefined value. The operation itself is undefined if there is insufficient stack space for the allocation. Allocated memory is automatically released when the function returns. The *AllocL* instruction is used to represent temporary variables that must have an address available. When the function returns, the memory is reclaimed. Allocating zero bytes is legal, but the returned pointer may not be unique.

ArgAddr

```
r1: 'type' = ArgAddr 'index'
```

ArgAddr represents Address of n-th argument. The address belongs to the stack part of memory. Takes two arguments: An integer index n specifies its position between all arguments handed to the function. A Type, so there is clear that if a structure is being passed, we can detect that the address of that structure is handed, but the necessary step to copy a structure needs to be done somewhere. One solution is to perform a deep copy while loading the address therefore the type is necessary. Another solution to this problem could be that Call instructions will copy the structure beforehand, pass the address as an argument, and then we only use this instruction simply as an address. The second solution is the one used for implementation.

Copy

```
r2: 'type' = Copy 'operand': 'type'
```

Copy is a simple instruction that represents a copy of another instruction respectively its value. This instruction is generated mostly by optimizations. But also can be useful in cases the value is reused and the target rewrites the contents of the first supplied register, which is also the case for tiny86. For example when using an argument address that cannot be easily counted as in the case of *AllocL* or *AllocG*.

ElemAddrOffset & *ElemAddrIndex*

```
r1:int=ElemAddrOffset 'base':int+'offset':int
r2:int=ElemAddrIndex 'base':int+'offset':int x'index':int
```

ElemAddrOffset and *ElemAddrIndex* are instructions for computing an address of a member in structure and the address of an element in an array. It performs address calculation only and does not access memory. The address is computed. ’ElemAddrOffset’ is used to get a member address inside the field. ‘ElemAddrIndex’ is used to get the address of n-th element at the ‘index’. The reason for implementing these instructions is purely academic, try to implement something unorthodox. They can be substituted with arithmetic operators, but they are handy when translation to IR is not aware of stack direction. Not sure if that could ever work.

Load

```
r1: 'type' = Load ['address']
```

Load is used to read from memory. After execution, the location of the memory addressed is loaded. If ‘address’ is not a well-defined value, the behavior is undefined.

Store

```
 Store ['address'] 'value':'valueType'
```

Store is used to write to memory. The result of a store is void ResultType because it doesn’t make sense to return a result from *Store* instruction. After execution, the contents of memory are updated to contain ‘value’ at the location specified by the ‘address’ operand. If ‘address’ is not a well-defined value, the behavior is undefined.

LoadImm

```
r1: 'type' = LoadImm 'value'
```

LoadImm loads ImmValue, which can be a type of double or int. This instruction represents a constant value in the program and resolves a type of value supplied.

C.4 I/O Instructions

GetChar

```
r1: int = GetChar
```

GetChar instruction represents a value of type char. value equals a single character that will be read from the console. If the value is used as an integer, then there will be an ASCII representation of that character. The same applies to PutChar instruction.

Putchar

```
PutChar 'value': int
```

PutChar takes one Instruction as a value to be printed to the console. Value needs to be a type of char. After execution, the char is printed to standard output.

C.5 Arithmetic Instructions

In this IR unlike in others, we specify *BinaryOp* and *UnaryOp* as single instructions with an argument that selects the operator. In most languages, every instruction is specified separately ex. ADD, SUB, . . . some also use different instructions for operating with Floats (FADD, FSUB, . . .). To reduce code in definitions in our IR we can specify only one *BinaryOp* itself, with an argument that is used to choose which operator this *BinaryOp* performs. Binary operators that are supported are:

Note: if not specified otherwise operation takes two integers or two floats.

- ADD – Addition,
- SUB – Subtraction,
- MOD, – Modulo operation, takes only two integers,
- MUL – Multiplication,
- DIV – Division,

- AND – Binary and, takes only integers,
- OR – Binary or, takes only integers,
- XOR – Binary exclusive or, takes only integers,
- LSH – Left shift, takes only integers,
- RSH – Right shift, takes only integers,
- NEQ – Not equal,
- EQ – Equal,
- LTE – Less than or equal,
- LT – Less than,
- GT – Greater than,
- GTE – Greater than or equal,

BinaryOp instruction takes BinaryOperator and two values with which the specified operation is performed. Value of a *BinaryOp* is the result of a specified arithmetic operation. The same applies to *UnaryOp*, except it takes only one value. Unary operators that are supported are:

- NEG – Logical negation,
- NOT – Unary complement (bit inversion), takes only an integer,
- INC – Increment,
- DEC – Decrement,

C.6 Conversion Instructions

Conversion Instructions serve a single purpose to safely convert a value from the IEEE representation used for the representation of floats to integers and backward.

Extend

```
r2:double = Extend 'value': int
```

The *Extend* instruction performs conversion of the value to the float type. Necessary for cast translation. Implicit conversion could be done by operators. But this is not the case for this implementation, all conversions are explicit, otherwise, operators don't pass assertion. The same applies to the next instruction.

Trunc

```
r1: int = Trunc 'value': double
```

The *Trunc* instruction performs conversion of the value to the integer type.

C.7 Other Instructions

Phi

```
r1: 'type'= Phi
    ['type', 'incBasicBlock' ]*
```

The *Phi* instruction is used to implement the Φ node in the SSA graph representing the function. There must be no non-phi instructions between the start of a Basic Block and the *Phi* instructions: i.e. Phi instructions must be first in a Basic Block. For the SSA form, the use of each incoming value is deemed to occur on the edge from the corresponding predecessor block to the current block At runtime, the *Phi* instruction logically takes on the value specified by the pair corresponding to the predecessor Basic Block that was executed just before the current block.

Appendix D

Contents of enclosed SD card

```
exe ........................................ the directory with executables
src ........................................ the directory of source codes
  thesis ............... the directory of LaTeX source codes of the thesis
  tiny-verse ............... the directory of the compiler source codes
text ........................................... the thesis text directory
  DP_Slavik_Martin.pdf ........................ the thesis text in PDF format
```