FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	x86-64 native backend for TinyC
Student:	Bc. Michal Vlasák
Supervisor:	Ing. Petr Máj
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2023/2024

Instructions

Analyze the current implementation of TinyC frontend used in the NI-GEN course. Determine any necessary changes and minimal viable runtime support for TinyC to be executed directly on the x86-64 architecture. Implement a compiler backend from TinyC IR used in the course to native machine code that demonstrates the use of advanced techniques mentioned in NI-GEN course for tasks such as register allocation and instruction selection. Implement a runtime based on your design that would allow TinyC programs on x86-64 to use system resources such as memory and I/O.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 5 December 2022 in Prague.

CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F8

Faculty of Information Technology
Department of Theoretical Computer Science

Master's Thesis

x86-64 native backend for TinyC

Bc. Michal Vlasák

June 29, 2023
Supervisor: Ing. Petr Máj

Acknowledgement

I wish to thank my supervisor, Ing. Petr Máj, for his patience and all the valuable advice he has given me.

I also wish to thank my parents for their support.

Declaration

In Prague on June 29, 2023

Michal Vlasák

Abstrakt / Abstract

Tato práce popisuje backend překladače, který kompiluje TinyC mezireprezentaci do nativních x86-64 instrukcí. Cílem bylo vytvořit překladač, který by ukázal složitosti spojené s architekturou x86-64, zvláště pak ve srovnání s architekturou Tiny86, kterou používají studenti předmětu NI-GEN.

V teoretické části je dán zvláštní zřetel na historii architektury x86-64 a pravou podstatu jejích omezení. Je vyhodnocen dosavadní výzkum v oblasti backendů překladačů a to s přihlédnutím k možnostem architektury x86-64.

Práce prezentuje návrh x86-64 TinyC backendu. Ten je založen na výběru instrukcí extenzivní peephole optimalizací a na alokaci registrů barvením grafů. Implementace samotná je důkladně rozebrána a mnoho praktických detailů je zváženo a vysvětleno.

Přeložené programy mohou buď využít systémový runtime jazyka C, nebo na Linuxu mohou být rozšířeny o vlastní minimalistický runtime dovolující programům běžet bez externích závislostí.

Text této práce a implementace jsou volně dostupné a mohou být použity nejen studenty předmětu NI-GEN.

Klíčová slova: překladač, backend překladače, TinyC, x86-64, výběr instrukcí, alokace registrů, C, běhové prostředí

Překlad titulu: x86-64 nativní backend pro TinyC

This thesis describes a compiler back end compiling TinyC IR to x86-64 native machine code. The goal was to create a compiler that would showcase additional difficulties imposed by the x86-64 architecture, especially compared to the simplified Tiny86 architecture targeted by students in the NI-GEN course.

In the theoretical part a close attention is paid to x86-64's history and the true nature of its limitations. Existing compiler back end research is surveyed and evaluated with respect to x86-64's capabilities.

A design for a x86-64 TinyC back end is presented. It is based on instruction selection by extensive peephole optimization and graph coloring register allocation. The implementation itself is presented thoroughly and many practical details are considered and explained.

The compiled programs are either able to use the system C runtime, or on Linux can be bundled with a custom minimalistic runtime allowing the programs to run without any dependencies.

The text of this thesis and the implementation are freely available, and can be used not only by NI-GEN students.

Keywords: compiler, compiler backend, TinyC, x86-64, instruction selection, register allocation, C, runtime

Contents /

Tables / Figures

Chapter 1 Introduction

Computers with hard-wired programs can serve only a specialized purpose and are not practical. For this reason computers accept input data as well as the *programs* to run. The programs are interpreted by the processor, and from the input data produce other data as output. Because most computers today are based on the binary numeral system (using only ones and zeros), the data as well as programs have to be encoded in *binary*.

Encoding machine instructions into binary by hand is tedious and error prone. Assembly language, a human readable textual notation of the instructions, came up as an abstraction and convenience for humans. However, still every computer or rather its CPU (*central processing unit*) can have different capabilities and support different instructions. Assembly doesn't abstract that—it is *machine specific*. High diversity was mainly true in the early days where every computer could be very different. But since then, the industry settled, and while CPUs can be still very different *internally*, they usually support one of the prominent *instruction set architectures*. The instruction set architecture (ISA) defines the interface of the processor—what instructions it supports and how they are encoded. A CPU implements that interface.

A more powerful abstraction for writing programs came with the advent of *programming languages*, which allow expressing programs at high level, without being tied to any machine or its capabilities. Programs written in programming languages are readable by humans, but not executable by machines. Programs usually have to be translated (*compiled*) to machine instructions, in order to run on the target machine.

Compilation can be a difficult process, because modern high level languages care to provide useful tools to the programmers, while machine instructions often support rather elementary operations. As there are many instruction set architectures and many programming languages, it may seem that a different compiler is needed for every combination.

In practice, most compilers today employ an architecture where the compilation is split into two steps:

1. *Front end*. Translates from human readable programming language to a programming language and machine independent *intermediate representation* (IR).
2. *Back end*. Translates from the intermediate representation to instructions of a particular instruction set architecture.

In this design, it suffices to have one front end for each source language and one back end for each target architecture. Compilation from any language with a front end to any machine with a back end is then possible. Additionally, operations that *any* compiler would want to do (due to being machine and programming language independent) can be performed on the intermediate representation in a common third part of a modern compiler:

3. *Middle end*. Performs programming language and machine independent transformations on the intermediate representation. Often these are *optimizations* that simplify or otherwise improve the code.

In the real world, there is still a place for newer or better compilers, that produce more efficient or smaller machine code. Not only are new programming languages created by the industry or academic research, but rarely even new instruction set architectures come up, or more often the existing architectures are extended with newer capabilities. Existing instruction sets are also implemented in newer and more modern processors, which may excel at slightly different operations and compiler writers often exploit even these implementation details, just to generate better code.

This means that even just from a back end perspective there is still a need for new compiler engineers. At the Faculty of Information Technology of Czech Technical University in Prague, there is a master's compiler course NI-GEN ("Code generation"), where students learn mainly about compiler back ends. The course features a programming language *TinyC* (a simplified version of the well known C programming language) and *Tiny86* (a simplified version of the widespread x86-64 architecture). Using simplified versions of a well known programming language and CPU architecture is great for education, because students are able to finish their own compiler in just one semester, but stay fairly close to the real world.

TinyC as a simplification of C is a rather low level language, which is expressive enough to be practical for humans writing production grade software, but it is low level enough that its concepts have relatively straightforward translations to machine code.

x86-64 although tracing its history almost 50 years back, can still in these days be considered the prevalent architecture. It poses many challenges not present in later designs, e.g. low number of registers, irregular register class hierarchy, complex addressing modes which are only partly regular, etc. As the original instruction set got extended throughout the years, some of the problematic areas were improved, but due to extensive backwards compatibility, we have to still account for them even today.

In this thesis our goal is to design and implement a back end compiling TinyC (or rather just the middle end IR used in the NI-GEN course) to x86-64. Such back end can be seen to accomplish several things:

- As a first back end to a real architecture it exercises the TinyC language and can show how practical it really is.
- It can show students, what difficulties a compiler faces when compiling to the real x86-64 architecture, instead of the simplified Tiny86.
- As the back end produces programs intended for an existing architecture, the produced programs can be inspected, debugged and measured with standard tools.
- We can compare the generated code to the output of existing C compilers.

Naturally, the back end will be usable by other front ends, so it is not strictly limited to TinyC.

Real world programs don't run in isolation. Apart from needing the processor to execute their machine code, they have other needs, that are towards the *runtime system* (which for compiled programs is usually the *operating system*). Programs want to allocate memory, perform I/O (input/output) operations, communicate with each other, run other programs, etc. Support for making these requests to the operating system is usually the job of a component called *runtime.* On the x86-64 the situation is no exception: there are instructions that allow the program to make requests to the operating system (*system calls*), but different operating systems allow system calls to be used differently and don't support the same features. The TinyC language doesn't nominally support calls into the operating system at all, and a *runtime* component that allows TinyC to use system resources has to be provided and made available to the TinyC language.

The following chapters provide a description of the x86-64 architecture and its challenges (chapter 2), a look of the state of the art in the area of compiler back ends (chapter 3), a close look at the design and implementation of an x86-64 back end for TinyC and the accompanying runtime (chapter 4) and evaluation accompanied with comparison to existing C compilers (chapter 5).

Chapter 2
x86-64 architecture

The x86-64 architecture, originally introduced by AMD under the name AMD64, and sometimes also called x64, is still one of the most prevalent architectures today. It is used in a most desktops, servers, but has also seen some use in mobile, low power, devices.

This widespread use of the architecture alone make it a worthwhile target for studying. From a compiler writer's perspective, it is interesting because of its CISC *Complex Instruction Set Computing* design, which is not straightforward to compile into.

For this reason the x86-64 has been used as an inspiration for Tiny x86 (also called Tiny86) designed by Strejc [Strejc, 2021], which is used for teaching purposes in the NI-GEN course at the Faculty of Information Technology CTU in Prague.

This chapter introduces the x86-64 architecture, its characteristics, constraints and compares it to the simplified Tiny86. Knowledge of some other instruction set architecture, or at least basic understanding of a processor, are expected, but familiarity with x86-64 is not.

This chapter takes the liberty of explaining some of the properties or advancements non-linearly with regards to the actual history, and sometimes tells little lies. From the perspective of x86-64, where all of gradual additions to the instructions sets are already available, it doesn't matter much, but note that some of the things presented, aren't really true for the actual 8086 16-bit processor. The Intel software developer's manual [Intel Software Developer's Manual, 2023] should be taken as the definitive resource, though we can wholeheartedly also recommend Volume 3 of AMD's Programmer's Manual [AMD Programmer's Manual Vol. 3, 2022] (which is focused on the general purpose instructions and their encodings and is sometimes more focused and little bit more approachable than Intel's equivalent), and the unofficial X86 Opcode and Instruction reference[1].

2.1 History

Although AMD introduced the x86-64 in the year 1999, the architecture has a much richer history. Interestingly enough, throughout the history, the instruction sets remained almost fully compatible. For this reason it is important to keep the history in mind when thinking about the x86-64.

The x86-64 is mostly still fully capable of behaving like its eldest ancestor, the Intel 8086, a 16-bit microprocessor introduced by Intel in 1978. This original design (loosely based on earlier Intel 8-bit processors) is where most of the instruction set originates.

The line of Intel 16-bit processors based on 8086 included 80186 and 80286, which added some new instructions and capabilities (like virtual memory). A big step was the 32-bit 80386 (also called i386), which extended most of the original 8086 design to 32-bits in a relatively straightforward way.

[1] `http://ref.x86asm.net/coder64.html`

After several iteration on i386, AMD introduced a 64-bit extension of the design, the x86-64 (also called AMD64), which also kept all the 32-bit additions by Intel, still staying fully backwards compatible. The extension to 64-bits was done in a different way then the extension to 32 bits and brought additional interesting limitations.

2.2 Characteristic

Due to the history of x86-64, it must be seen as a 16-bit processor operating in a 32-bit mode by default and additionally supporting 64-bit operations. In this section we introduce some of the main characteristics of the original 8086 and later see how they are reflected in the x86-64.

2.2.1 Basics

Like most CPU architectures these days, x86-64 is *register* based. This means that interface of the CPU consists of a fixed number of registers and instructions that allow operations on these registers. We can describe instructions on the x86-64 in textual assembly syntax, which lists the *mnemonic* and the *operands* of the instruction:

```
⟨mnemonic⟩ ⟨op1⟩, ⟨op2⟩, ⟨...⟩
```

Most instructions then read some of the operands (usually registers) and write into other operands (also usually registers). The instructions usually represent simple arithmetic operations, but also more complex instructions are available. A concrete example of an instruction is addition of two registers `ax` and `bx`:

```
add ax, bx
```

This instructions reads the contents of the `ax` and `bx` registers, adds them together and writes the result to the `ax` register. Most instruction 8086 arithmetic instructions work this way—they have two operands, both are used as sources and the first as the destination for the result. We call this *two address code*, as opposed to *three address code*, which has two sources and one destination.

Most operations manipulate registers and values stored in memory. For example, there is an instruction to copy ("move") a value from one register to another:

```
mov ⟨reg⟩, ⟨reg⟩
```

But similar instructions that address memory locations do slightly different things:

```
mov ⟨mem⟩, ⟨reg⟩
mov ⟨reg⟩, ⟨mem⟩
```

In the first instruction, we copy a value from register to memory, and with the second instruction we copy a value from memory to a register. We can call these *store* and *load* instructions. Even though they use the same mnemonic (`mov`), they do different operations and they are encoded a bit differently. We can view them as distinct *instructions*. Internally, what distinguishes them is an opcode—a number, which uniquely identifies all instructions and the kinds of operands. Together, we will call the kinds of operands allowed by an instruction an *addressing mode*. Due to details that will become clear soon, 8086 instructions mostly allow two kinds of operands:

- a register, ⟨*reg*⟩
- a register or memory location ⟨*reg/mem*⟩

For this reason register copy, load and store instructions correspond to just two *opcodes*, which cover all three combinations:

```
mov ⟨reg/mem⟩, ⟨reg⟩
mov ⟨reg⟩, ⟨reg/mem⟩
```

Two different opcodes can actually encode a register to register copy operation and we can choose arbitrarily.

With the nomenclature in mind, we can for example call the following:

```
mov ax, [bx]
```

a load instruction, but is actually no more than an instruction with opcode (137 or 10001001 in binary), which uniquely encodes both the semantics (copy from second operand to first) and the addressing mode (first ⟨*reg*⟩, second ⟨*reg/mem*⟩).

The distinction between opcode and mnemonic is important, because in a compiler we mostly care about the semantics and the addressing mode of the operation (i.e. the opcode and the actual operands), not how it is written textually in assembly, which is just a representation that is meant to be convenient for humans. The difference can be more subtle for some mnemonics like `imul` which have a few very different forms.

2.2.2 Registers

There are 8 registers on the 8086. They have names and some are sometimes used in special ways. But mostly, we can call the registers *general purpose*, since usual operations all work with them and allow either register to be used.

The full registers (like `ax`) are 16-bit, but separate access to either the upper 8-bits or the lower 8-bits is allowed through the `ah` ("`ax` high") and `al` ("`ax` low") registers. Because of this, registers like `ah` and `al` are not real registers, but just different names for parts of the existing registers.

Not all registers allow access to the 8-bit parts. The registers that are only accessible as 16-bit are `sp` (*stack pointer*), `bp` (*base pointer*, also known as frame pointer), `di` (destination index) and `si` (source index). As can be seen from their names, they are often used for special purposes and hold potentially large integers or addresses, which need the full 16 bits. On the other hand, the other four registers are known as the *accumulator* registers—they are just general purpose registers intended for most arithmetic and called in the alphabetical order `ax`, `bx`, `cx`, `dx`. All registers are still general purpose, the categories just signify how they are *usually* used.

`bp` and `sp` are usually used for special purpose—as pointers to the stack. The `sp` instruction points to the top of the system stack (growing towards lower address), which is also used as the call stack, and `bp` points to the *base* of the current function's stack frame. As function calls are nested, return addresses and base pointers of the previous functions are pushed to the stack and popped back on returns.

It is no coincidence that there are eight 16-bit registers and eight 8-bit registers. Most things on the 8086 come in eights. This stems from the actual encoding of the instructions, where a register is encoded with 3 bits. In a context where 16-bit register is expected, these 3 bits as a number identify one of the eight 16-bit registers, and in 8-bit contexts the 3 bits mean one of the halves of the four accumulator registers.

Most instructions have two variants: 8-bit and 16-bit which work with the respective registers, or *byte* (8-bit) and *word* (16-bit) memory locations in memory.

For example an 8-bit variant of a load instruction has opcode 136 (16-bit variant is 137) and can look like this:

```
mov al, byte [bx]
```

There are two important things to note: only the first register became 8-bit, the memory location is still identified by a 16-bit pointer. Also, in the textual assembly, we can specify the memory location's operand size with either `byte` or `word`. The keyword is mostly optional if the operand size can be deducted otherwise (like here from the 8-bit `al` register).

2.2.3 Memory locations

We have already seen, that there are addressing modes which allow memory locations to be encoded. These memory locations allow values to be stored to memory or loaded back. But a lot of other instructions allow their operands to be values stored at the memory locations. We usually write memory locations in between brackets (`[`, `]`) to distinguish them from ordinary uses of registers.

In general, memory locations are either directly specified with an integer literal (*an immediate*) or computed from values of registers and immediates. On the x86-64 (not on earlier iterations of the instruction set), there are two main addressing modes:

1. *SIB mode*, where SIB stands for *scale*, *index* and *base*. The memory locations are in the following form:

   ```
   [⟨base⟩ + ⟨scale⟩ * ⟨index⟩ + ⟨displacement⟩]
   ```

   Base and index are registers, scale is either 1, 2, 4 or 8 and displacement is an immediate value (on 8086 either 8-bit or 16-bit). All of the individual parts are optional. All of the following are valid:

   ```
   [100]
   [ax]
   [ax + 100]
   [ax + bx]
   [ax + 2 * bx]
   [ax + 8 * bx]
   [ax + 8 * bx + 100]
   [8 * bx]
   [1 * bx + 100]
   ```

2. *RIP-relative mode*. This encoding addresses relatively to the current *instruction pointer* (on 8086 called `ip`). On x86-64, the instruction pointer points to the beginning of the *next* instruction, not to the current one. Addressing relative to the instruction pointer is new addition of x86-64. The instruction pointer is not necessarily a real register, but this addressing mode acts as if it was. We can imagine the mode as if the following was allowed:

   ```
   [ip + ⟨displacement⟩]
   ```

   A displacement (relative offset) is added to the instruction pointer. This is useful for *position independent code*, often used for libraries, because as long as the whole library is in a continuous piece of memory and uses RIP-relative addressing, it can actually be anywhere in memory. This is unlike absolute addresses, which require so called *relocations*.

   Due to the relative addressing nature, assemblers usually use a different syntax for RIP-relative addressing, e.g. for NASM:

   ```
   [rel ⟨label⟩ + ⟨constant⟩]
   ```

   Assembler will transform this to the offset to ⟨*label*⟩ + ⟨*constant*⟩, relative to the next instruction.

2.2.4 Arithmetic

As with registers, even arithmetic operations on the x86-64 come in eights. There are:

- 8 binary operations (we will sometimes call them *group 1*),
- 8 shift operations (*group 2*)
- 8 unary operations (*group 3*)

Operations in these groups have the same encoding (the real reason for the grouping) and often also very similar nature. In no category are all the 8 operations useful for translating usual programs, but these are some notable examples of instructions in these categories:

- Binary operations: `add`, `sub`, `and`, `or`, `xor`.
- Shifts: `shl` (shift left), `shr` (shift logical right), `sar` (shift arithmetical right)
- Unary operations: `not` (bitwise complement), `neg` (two's complement negation), `mul` (unsigned long multiplication), `imul` (signed long multiplication), `div` (unsigned long division).

Binary arithmetic is straightforward and except for working with registers, it allows a memory location as either one of the two operands (but not both), e.g. the `add` operation has the following modes:

```
add ⟨reg/mem16⟩, ⟨reg16⟩
add ⟨reg16⟩, ⟨reg/mem16⟩
add ⟨reg/mem8⟩, ⟨reg8⟩
add ⟨reg8⟩, ⟨reg/mem8⟩
```

Additionally, an immediate value can be used as the second operand:

```
add ⟨reg/mem8⟩, ⟨imm8⟩
add ⟨reg/mem16⟩, ⟨imm16⟩
add ⟨reg/mem16⟩, ⟨imm8⟩
```

In concrete examples:

```
add ax, bx
add [ax], cx
add [ax+20], cx
add dx, [ax+2*bx+20]
add dx, 1
add [ax], -5
```

There are two immediate sizes available for 16 bit operations—the 8-bit one can save a byte if the immediate is small. In cases when the immediate is smaller then the operand size, it is *sign extended* to the operand size. x86-64 uses two's complement representation for negative numbers.

There isn't a separate signed and unsigned version of `add` and some other instructions. This is because the produced result is the same for both signed and unsigned operations—they differ only in how the bits are *interpreted* by later operations and how an invalid result is detected (either with the carry out bit, or with the overflow bit). Carry and overflow are made available through the special `flags` register, which has a bit for each of the flags (like carry, overflow, but also the zero flag, parity flag or sign flag). Every arithmetic operation sets the flags register based on the flags of the result of the arithmetic operation.

Unary operation like bitwise complement (also one's complement negation) have the following modes (note that they use the source also as destination):

```
neg ⟨reg/mem16⟩
neg ⟨reg/mem8⟩
```

One operand is necessarily not enough for long multiplication and division which are also encoded like unary operations. They are also not really binary operations—they are *long* multiplication and division, i.e. at least one of the operands is twice as large (i.e. 32-bit on 8086), which needs *two registers*. These two registers are implicitly understood to be `ax` and `dx`. With multiplication, `ax` is multiplied by the single operand and the result put into `ax` (low 16-bits of the result) and `dx` (high 16-bits of the result). With division, 32-bit value whose upper 16-bits are in `dx` and lower 16-bits in `ax`, is divided by the single operand and the quotient is put into `ax` and the remainder into `dx`.

Ordinary multiplication, where the result is limited to the size of the inputs is possible with the `imul` instruction. Since it produces 16-bit result from two 16-bit operands, it can actually be used for both signed and unsigned multiplication. This instruction is much more suitable for compiler generated code, which often can't make use of the high 16-bits of the 32-bit result anyways.

Shifts are similarly restricted. They are encoded as unary operations and there are only these modes as demonstrated on `shl`:

```
shl ⟨reg/mem16⟩, imm8
shl ⟨reg/mem8⟩, imm8
shl ⟨reg/mem16⟩, 1
shl ⟨reg/mem8⟩, 1
shl ⟨reg/mem16⟩, cl
shl ⟨reg/mem8⟩, cl
```

I.e. either the opcode specifies the shift by one implicitly, or there is an immediate value. Shifts by variable amount have to store the shift amount in the `cl` register (lower 8-bits of `cx`). 8-bit shift amount is enough even on 64-bit machines where the maximum shift amount is 63—the processor actually masks this and limits the maximum shift amount.

2.2.5 Condition codes

As mentioned, arithmetic operations set flags. These flags can be used for conditional execution. The flags register can be interpreted in one of the 16 ways, e.g.:

- *Zero* (`z`). Is true if the zero flag is 1.
- *Not zero* (`nz`). Is true if the zero flag is 0.
- *Equal* (`e`). Is true if the zero flag is 1.
- *Less than* (`l`). Is true if sign flag is not equal to overflow flag.
- *Below* (`l`). Is true if carry flag is 1.
- ...

While there are 16 predetermined ways to interpret the flags, we use more than 16 names for these condition codes. For example, to detect that the value is zero, the zero flag can be checked to be 1 through the `z` condition code. But the same condition code checking zero flag to be 1 can be used for equality checks—values are compared by subtraction and if the result is zero, then the values are equal. But in that case we usually call the condition code `e` ("equal"). Multiple *semantic mnemonics* compile to just 16 condition codes.

Another thing of note is that there are two ways of doing a "smaller" check—*less than* and *below*. The first is intended for unsigned numbers (and uses the carry flag) and the second is for signed numbers (and checks the carry flag *and* the overflow flag). These are one of the operations where signedness matters.

The following instructions have condition codes as part of their opcodes (`cc` stands for the condition code):

- `jcc` ⟨*destination*⟩—jumps to destination if the condition is true. E.g.

  ```
  jz label
  jnz ax
  jl [ax]
  ```

- `cmovcc` ⟨*reg16*⟩, ⟨*mem/reg16*⟩—copies source to register destination, but only if the condition is true. E.g.

  ```
  cmove ax, bx
  cmovl ax, [bx]
  ```

- `setcc` ⟨*reg/mem8*⟩—sets lower 8 bits to 00000001 if the condition is true, and to zero otherwise.

  ```
  setl al
  sete ah
  setz [ax]
  ```

The condition code (4 bits) is part of the opcode for these instructions.

The `jcc` jump instruction is the alternative to the unconditional `jmp` instruction. The `call` instruction has the same addressing modes as `jmp`:

```
jmp ⟨imm16⟩
jmp ⟨imm8⟩
jmp ⟨reg/mem16⟩
```

All three instructions allow instructions allow relative jumps based on immediates, but only `jmp` and `call` allow absolute jump addresses to be read either from register or from memory.

2.2.6 Encoding

All of the opcodes, addressing modes and sizes of operands, are deeply connected with the actual encoding of the instructions. On the x86, most instructions follow this scheme:

1. *Prefix bytes.* Optional, one byte per prefix. Some instructions allow behavior to be changed by a prefix, e.g. the `lock` prefix makes some arithmetic instructions atomic.
2. *Opcode byte.* Specifies the opcode, i.e. both the operation and the addressing mode.
3. *ModRM byte.* Encodes two operands, ⟨*reg*⟩ and ⟨*reg/mem*⟩.
4. *SIB byte.* Encodes the memory location (see section 2.2.3).
5. *Immediate value bytes.* Any offsets, displacements or other immediates.

The ModRM byte is the most important thing when considering addressing modes. It is present for most instructions (only e.g. instructions without any operands don't use it). It encodes up to two operands, one register, the other either register or memory location. The name of the ModRM byte stems from its three components:

- *Mode.* 2 bits.
- *Register.* 3 bits.
- *Memory or register.* 3 bits.

The four modes are able to encode the following pairs of operands:

```
⟨reg⟩, [⟨reg⟩]
⟨reg⟩, [⟨reg⟩+⟨disp8⟩]
⟨reg⟩, [⟨reg⟩+⟨disp16⟩]
⟨reg⟩, ⟨reg⟩
```

One operand is always a register, the other can be either a register or a memory location given by register and an optional big or small displacement. This is a form of more compact encoding for the common case where the memory location is given just by a register. It fits one special case of the SIB memory. The others have to be encoded with a separate SIB byte, which has the following components:

- *Scale.* 2 bits. The actual scale is 2^{scale}.
- *Index.* 3 bits.
- *Base.* 3 bits.

The encoding is fairly straightforward. Though as presented has a few problems: none of the ModRM modes uses the values from SIB, and in SIB absence of index or base cannot be encoded in 3 bits if the registers are also 3 bits. Unfortunately, there are *special cases* on the x86-64. They are too confusing to list here, but they can be found in [Intel Software Developer's Manual, 2023]. Simply put, usually one of the possible values of a field is reserved for special meaning (e.g. `sp` in the `M` part of ModRM actually means that SIB should be present and consulted). There are necessarily a few unencodable memory locations, but there is usually an alternative way of specifying the same (e.g. with a displacement of 0). The only exception is using the `sp` stack pointer as an index register, which is not encodable. Fortunately, using `sp` as index register would rarely make sense, and it can still be used as base, which is more useful.

It is important to note that the ModRM byte always encodes the two operands in the same way. The meaning and order of the operands is determined from the opcode. For example, most binary operations allow ⟨*reg/mem*⟩ as either the source or the destination. The direction is usually specified in one bit across all binary operation opcodes, we can call it the *direction bit*.

Similarly, the operand size is also determined by the opcode and is one of the bits of the opcode. Zero in the *operand size bit* means 8-bit operation, one means 16-bit operation.

For operation which encode only one operand (like shifts, `neg` or `idiv`), only the M operand is used. Because of this, the three bits for the R operand are free and instead used for extension of the opcode. It is no coincidence that there are 8 shifts and 8 unary operations, because the exact kind of the operation is encoded in these three bits. For example, opcode 192 (`0xC0`) corresponds to shifts in the following addressing mode:

```
⟨shift⟩ ⟨reg/mem8⟩, ⟨imm8⟩
```

The exact kind of shift (`shl`, etc.) is encoded as if it was the ⟨*reg*⟩ parameter unused by this addressing mode.

Interesting with regards to encoding is the `lea` (*load effective address*) instruction:

```
lea ⟨reg16⟩, [⟨mem⟩]
```

It is encoded naturally with the ModRM and SIB bytes. But instead of performing anything with the value at the memory location, the instruction just computes the address and stores it to the register. This is not only useful for calculating addresses that can be reused several times (instead of being recomputed in every time), but can also be used for ordinary arithmetic: memory locations are formed with addition and simple multiplication (shift). Though `lea` doesn't set flags, unlike other ordinary arithmetic instructions, so it is not a universal replacement for addition.

2.3 32-bit extension

In the previous sections we introduced features of 8086 and later additions (x86, x86-64), *as if* they were supported by the original 16-bit processor. Often they are not, but by consistently introducing everything like it would be on the 16-bit processor, we can look at how the 32-bit extension was done by Intel.

The extension was straightforward: the processors came with two modes—32-bit, but also a 16-bit mode for backwards compatibility. In the new 32-bit mode, everything 16-bit is now 32-bit, including operand sizes, immediate sizes, displacements, etc. This meant that in 32-bit the two supported operand sizes are 8-bit and 32-bit and these are distinguished by the opcode. The 32-bit operations are encoded exactly like the 16-bit processors, but with larger immediates.

The processor is thus capable of doing both 16-bit operations and 32-bit operations, but they are multiplexed in the same opcodes. To allow 16-bit mode to use 32-bit operations and vice versa, the *operand-size prefix* (`0x66`) was introduced. 8-bit operations have separate opcodes, so these are available in both modes.

Similar prefix was introduced for addresses—the *address-size* prefix, which is able to make a memory location be based on 16-bit calculations on 32-bit systems and vice versa. Though this is less useful in practice.

All of the registers got extended to 32-bits. The full 32-bits became available with the `e` ("extended") prefix of the register name, e.g. `eax` or `esp`. The upper 16-bits of the registers are no longer addressable individually and can change only as part of the full 32-bit operations.

2.4 64-bit extension

When AMD extended x86 to 64-bits they brought also many other extensions. Notably this included 8 additional registers (with no descriptive names, just numbered from 8 to 15) and the extension of registers to 64 bitt (the full 64 bits are accessible with prefix `r` instead of `e`, e.g. `rax` or `rsp`). The encoding had only 3 bits available to encode a register, so a new mode or operand size prefix was not enough, especially since backwards compatibility was still desired.

The solution was the REX ("register extension") prefix. It is a byte, where the 4 most significant bits are constant (0100, i.e. 4) and the 4 individual least significant bits have individual meaning as follows:

- REX.R extends the *R* field of ModRM to 4 bits.
- REX.X extends the index of SIB to 4 bits.
- REX.B extends the base of SIB to 4 bits.
- REX.W specifies operand size to be 64-bit.

The REX byte actually fulfills multiple purposes—it makes 8 additional registers encodable for any instruction, because all use the ModRM (and SIB bytes) to specify the operands. Additionally it makes available the 64-bit operand size and is like the operand-size prefix.

The reason why 64-bit operand size has to be requested explicitly is, that the default operand size on x86-64 is still 32-bits. Either 8-bit or 32-bit opcodes are available by default. REX byte has to be applied before the 32-bit opcodes to achieve 64-bit operand size.

Like operand size, also the immediate size stayed at 32 bits. Only a few instructions were allowed full 64-bit immediates. The most notable one is:

```
mov ⟨reg64⟩, ⟨imm64⟩
```

which allows 64-bit immediates to be loaded into a register directly, without bit twiddling. 64-bit immediates were disallowed likely because they are both too large, which negatively impacts the code size, but also because they are mostly unnecessary—most useful immediates are “small” positive and negative numbers, which can still be achieved through the implicit sign extension of the 32-bit or even 8-bit immediate.

Another important change is, that operations with the 64-bit operand size automatically zero out the upper 32-bits of the 64-bit registers. This is different from 16-bit operations, which don’t change the upper 16 bits of the lower 32-bits. This behavior implies that 32-bit operations don’t care about the upper 32-bits of the 64 bit registers, which is actually very important for modern super scalar processors. If, with the default 32-bit operand size, the 32-bit operations only changed parts of the registers, the operations would *depend* on the unchanged upper 32-bits. These kinds of *partial dependencies* are detrimental for performance, because they are mostly false dependencies, but still have to be obeyed by the processor.

For this reason, on 64-bit systems the 32-bit operations are very efficient, since they are narrower than 64-bits, but still don’t have problem with partial dependencies as operations on 8-bit or 16-bit portions of the registers do.

If a 32-bit operation doesn’t need to use any of the 8 new registers, it can also spare the REX prefix. On the other hand, 64-bit operations already require the REX prefix, so there is no penalty in using all of the available registers.

When *any* REX prefix is used, then the 8-bit registers change behavior—instead of addressing lowest and second lowest 8-bits of the first 4 registers, they will address lowest 8-bits of all registers. This makes x86-64 a bit more regular then its predecessors, because no longer is there a distinction between accumulator and other registers, provided are willing to in some cases use an otherwise unneeded REX prefix byte.

2.5 Calling conventions

As functions are called, there has to be an agreement in how resources are shared between the calling function (*caller*) and the called function (*callee*). An ABI specifies this as well as the details on how arguments are passed, what layout does the stack have, how the functions are named and how big are essential types. For the x86-64 architecture, there are two prevalent ABIs:

- System V AMD64 ABI [System V AMD64 ABI, 2023]. Used on most operating systems.
- Microsoft x64 calling convention [Microsoft x64 calling convention]. Used on Microsoft Windows.

The calling conventions share the definition of some C types like `char` (1 byte), `short` (2 bytes) and `int` (4 bytes), but disagree for example on `long` (8 vs 4 bytes).

Both calling conventions use registers for passing arguments and return values, but disagree on the registers themselves. Callee saved and caller saved registers also differ (these are explained in more detail in section 3.8.3.6). Even alignment of stack is handled differently.

From a perspective of a compiler, the system ABI is not important to *internal functions*, which are only seen by one compiler, but functions which are either called by external code, or which call external code must respect the system ABI. The ABI provides standard interface, and with that even code from multiple compilers can be linked or even dynamically linked together and run without problems.

Languages like C usually abide the system ABI's calling conventions for *all* functions. In any case, functions from the C standard library usually are based on the C ABI, so at least calls to standard functions have to respect the ABI.

2.6 Operating system interface

On the x86-64, programs ultimately use the `syscall` instruction to transfer the control to the operating system. The instruction itself saves the instruction pointer (the return address) to `rcx` and `rflags` to `r11`. Anything else is up to the operating system and thus subject to a calling convention of its choosing. If this system call calling convention is documented for applications, then applications can use it and call into the OS directly.

Linux is one of the few operating systems that documents the calling convention used for system calls. The convention is very similar to the C one specified by the System V ABI (they are in fact both specified in the same document [System V AMD64 ABI, 2023]). Though for system calls, an extra integer is passed to hold the system call number (because the `syscall` instruction doesn't specify "what to call" in any way, unlike a normal `call` instruction).

On other operating systems like OpenBSD and Solaris (using the System V ABI for ordinary calls), or Microsoft Windows, the system calls and the system call calling convention are not specified and no guarantees are made about their stability from version to version. On these operating systems usually the system C library abstracts requests to the operating system.

We can distinguish between the *standard part* of the system C library and the OS specific *system part*. By making the operating system interface the system C functions, the operating system is free to only abide to the normal C calling convention and may choose to implement simple system calls without a `syscall` if a switch to kernel space is not needed.

2.7 Comparison with Tiny86

Tiny86 [Strejc, 2021] is a 64-bit architecture inspired by x86-64. It copies many of the design decisions of x86-64, which is great because it is familiar (FIT students encounter x86-64 in other courses), non-trivial as a target, and still popular in practice.

2.7.1 Registers

One of the main differences are the available registers. On x86-64 there are 16 *general purpose* registers, of which two are usually used for special purposes (`rbp`, `rsp`) and some *special purpose* registers like `rflags` or `rip`, which are only accessible through "special" instructions. Tiny86 separates these 4 special registers and on top allows configurable number of 64-bit general purpose and 64-bit floating point (i.e. double precision) registers. No access to narrower parts of the registers is allowed.

2.7.2 Operations

Tiny86 supports pretty much the same set of arithmetic operations the x86-64 does. Notably two separate instructions are available for computing the remainder (confusingly called `mod`) and quotient. There isn't long multiplication (i.e. multiplication of two 64-bit numbers only gives 64-bit result). This makes it much easier to use the instructions, as it doesn't involve any implicit operands.

Operations that are distinct in signed and unsigned variants are also distinguished on Tiny86, just like on x86-64. Unfortunately, logical right shift (unsigned version of the right shift) is not present, while the arithmetic right shift (the signed version) is. This makes it harder for unsigned operations to be lowered.

Special purpose instructions are available for debugging, as well as for reading numbers from standard input and for writing numbers to standard output. These are implemented by the Tiny86 VM with calls to the C++ standard library.

2.7.3 Addressing modes

The following addressing modes are supported by Tiny86:

- For binary operations:

```
⟨binop⟩ ⟨reg⟩ ⟨reg⟩
⟨binop⟩ ⟨reg⟩ ⟨imm⟩
⟨binop⟩ ⟨reg⟩ ⟨reg⟩ + ⟨imm⟩
⟨binop⟩ ⟨reg⟩ [⟨reg⟩]
⟨binop⟩ ⟨reg⟩ [⟨imm⟩]
⟨binop⟩ ⟨reg⟩ [⟨reg⟩ + ⟨imm⟩]
```

- For unary operations:

```
⟨unop⟩ ⟨reg⟩
```

- For move and `lea` instructions:

```
mov ⟨reg⟩, ⟨reg⟩
mov ⟨reg⟩, ⟨mem⟩
mov ⟨reg⟩, ⟨imm⟩
mov ⟨reg⟩, ⟨reg⟩ + ⟨imm⟩
mov ⟨mem⟩, ⟨reg⟩
mov ⟨mem⟩, ⟨imm⟩

lea ⟨reg⟩, ⟨mem⟩
```

  where ⟨*mem*⟩ is the following (all components are optional):

```
[⟨reg⟩ + ⟨reg⟩ * ⟨imm⟩ + ⟨imm⟩]
```

The interesting thing is, that the addressing modes for `mov` and `lea` instructions are a superset of the SIB mode on x86-64. Like all immediates on Tiny86, even those specifying memory locations are 64-bit. This makes the instructions much more flexible in access to array elements than x86-64, which allows the scale to be only a small power of two.

Unfortunately, unlike x86-64, Tiny86 chooses to allow different memory accesses in binary operations and to not allow memory access for unary operations. These make the instruction set rather irregular, especially since the destination is always limited to a register.

The unsigned limitations and support for only 64-bit operations is not a problem, since the TinyC language supports exactly those.

2.8 Conclusion

The addressing modes on x86-64 are fairly regular, even if it may not seem so. The actual encoding is more complex because of the special cases, but all useful operand combinations are allowed, so the complexity is limited to the encoder/decoder, and not the compiler, which sees the operations as fairly orthogonal.

Tiny86 is much more regular in some regards, but less so in others. The main advantage of x86-64 is the memory location specifications, which are the same for all instructions. This is not true on Tiny86 and can even make compilation with some techniques harder.

The x86-64 architecture is regarded as a CISC architecture, but in practice we see it more as an architecture aiming for compact instruction encoding. Some other CISC designs, like the VAX, make very different decisions and have fully orthogonal addressing modes, where the kind of each operand is encoded separately and not as part of the opcode.

x86-64 on the other hand, is only regular in its irregularities. This actually makes it a very interesting compilation target, as it has the complex CISC features, but also the irregularities, which test compiler capabilities.

Chapter 3
State of the art

This chapter's goal is to introduce and evaluate techniques and approaches that are already known and described in literature. We mainly focus on parts of the compiler that are part of the back end, or very much connected to it.

3.1 Structure of a compiler backend

A compiler is a tool that transforms the source language into the target language. Doing so *directly* is however hard as both the input and output language may be fairly complex and in very different, incompatible ways.

Like with many things, an extra layer of indirection helps, and compilers got split into more and more individual components.

Most compilers today are split into at least three parts: front end, middle end an back end. This separation is very useful not only to solve the $m \cdot n$ compilers problem, but allows each part to stay more focused.

A compiler back end is no exception. Translating directly from a machine independent IR into instruction set (like x86-64) is hard. The classic architecture used by countless compilers are the following three phases:

- *Instruction selection.* Chooses *which* machine instructions to use.
- *Instruction scheduling.* Choose the *order* of the instructions.
- *Register allocation.* Allocates and assigns machine registers to instructions.

This sequence is applicable to register based machines—the instructions have to already be assigned some registers during instruction selection. But it is much more convenient to pretend to have infinite amount of registers. This way only the late register allocation stage has to be concerned with limited amount of *physical registers*. The earlier stages may instead use arbitrary amount of *virtual registers*.

Even splitting into these or more stages doesn't fully help with the complexity—most of formulations of all three stages are NP-complete [Cooper et al., 2004].

Another important component to consider with regards to the back end is the middle end. It is the middle end, which provides input to the back end. Either instruction selection works directly on the middle end IR, or has its own IR somebody needs to translate to.

The middle end can influence the back end greatly. Especially if SSA form (see section 3.4) is used in the middle end. The SSA form has to usually be deconstructed at some point, and the back end is usually the more suitable as it doesn't *need* the SSA form (although approaches exist to leverage SSA form for example in register allocation, see 3.8.4.4), while middle end usually finds it much more useful and doesn't have a reason to deconstruct it early.

3.1.1 Phase ordering

A problem that will become clear gradually deserves a mention up front. There is a problem with the classic three stages in a back end. The order in which they are is not ideal. But in fact, no order is ideal, the phases all depend on each other: [Cooper et al., 2004].

- Usually not all registers are equal, and instructions sometimes have specific register constraints. Registers cannot be allocated until it is known what and *how many* registers are needed.
- Different instructions allow very different schedules. But different schedules might make different instructions more plausible or interesting (e.g. due to differences in execution unit saturation, see section 3.7).
- Insufficient amount of physical registers may result in need to store values in memory. This requires new instructions that should be selected and scheduled.

The classic order of instruction selection, instruction scheduling and register allocation turns out to be the best compromise. Still, register allocation may not only create new instructions, but also delete others (see section 3.8.3.4). Not only for this reason there is often a fourth, last, component of a back end: the *peephole optimizer*.

We will introduce all these four main components as well as SSA and SSA-destruction in the following sections

3.2 Intermediate representations

The intermediate representation (or more of them if the compiler is spit into even more stages) is the cornerstone of the compiler. Transformations on the middle end IR work for any source language and for any target language. Due to this, the intermediate representation should be made to suite the intended transformations.

3.2.1 Abstract syntax tree

Source languages (like TinyC) are sequences of characters. As they get parsed, a parse tree is implicitly created. Often though, the more interesting representation is the Abstract syntax tree (*AST*).

The AST can also be regarded and used as an intermediate representation in the compiler. However, it has a couple of disadvantages. Mostly, it reflects the *syntax*. It is too close to the source language to be suitable as a language independent middle end IR.

Still many operations can be done just fine on trees. They are simple to process and modify. A language independent tree representation is a perfectly valid representation.

3.2.2 Directed acyclic graph

Often trees become too limiting, especially for some optimizations like *common subexpression elimination* (CSE). If our program is something like:

```
(a + b) * (a + b)
```

then we would like to notice and use the fact that the operands of the multiplication are in fact the same. However, if we were to reuse the first `(a + b)` node, it would have “two parents” (the multiplication operation twice). This is not possible in trees, but is possible with *directed acyclic graphs* (DAGs). DAGs are very natural especially for common subexpression elimination or value numbering optimizations.

3.2.3 Three address code

Three address code can be seen as a linearization of a DAG [Aho et al., 2006]. For example, the above could be rewritten with the common subexpression eliminated as follows:

```
t1 = a + b
t2 = t1 * t1
```

The results are given explicit names and form *temporaries*. The name *three address code* stems from the three variables involved in the operation—one is the destination, and the other two are sources. Unlike with DAGs, the operations are ordered. While DAG captures *data-flow* (how operands from to operations). On the other hand if we assume the operands in three address code to be variables, three address code doesn't really capture data-flow, as the variables are assigned and read. But three address code captures *control flow*. If however, the three address code operands represent the values directly, and the names on the left side are merely labels naming the values, we have a representation which captures both data-flow and control flow (see also 3.4.1).

3.2.4 Control flow graph

Three address code operations can be ordered linearly. Then the control flow is either continuing from one triplet to the next, unless a special jump operation jumps to a different part of the sequence.

Usually a more exact representation of control flow is desired. One where explicitly each node has links to its successors and predecessors. This forms a *control flow graph*, where edges are possible transfers of control flow and nodes are triplets. Naturally, cycles in programs manifest as cycles in the control flow graph, etc., so here we have a true general graph.

But most triplets (except for jumps and conditional jumps) continue explicitly to the next triplet. To prevent expensive representations of all edges, we can make most of them implicit by grouping into (maximal) sequences of triplets, which are only entered at the first triplet and left with the last triplet. We call these *basic blocks*. Inside basic blocks, the control flow edges are implicit, but among basic blocks, the control flow edges have to be represented explicitly.

3.3 Peephole optimization

Peephole optimization is a general technique that examines a small part of a program (the *peephole*), in which instructions are examined and optimized according to known (or in other way derived) patterns.

Peephole optimization is applicable to any intermediate representation, but is especially effective on data-flow graphs, where it is able to investigate *logically* adjacent operations, but is also often used on linear representations, where it examines *physically* adjacent instructions.

Optimizations are possible even with patterns as small as one triplet. E.g. the following:

```
t1 = t2 * 8
```

can be optimized to:

```
t1 = t2 << 3
```

3.4 SSA form

SSA (*static single assignment*) form is a form used by most of today's optimizing compilers, including for example LLVM[1] and GCC[2]. It simplifies and makes faster a lot of classic optimizations.

Like the name suggests, static single assignment form stands on the fact that each variable is assigned exactly once. On top of that, we are interested only in *static* assignments, meaning, that there is only one program point that assigns a variable—contrary to *dynamic* assignments, which would count how many times the assignment is executed at runtime, i.e. how many times the execution gets to the single program point which assigns the variable.

There are many important benefits to SSA form, of which we highlight a few:

- Since each variable is assigned only once, no variable is ever reassigned. Thus value held by the variable is always available. This means that algorithms don't have to be cautious about using a definition that is reassigned.
- There is no ambiguity in what *definition* of a variable a *use* can refer to, since each variable has exactly one definition. This means that instead of using *use-def chains* (a data-structure that links together all definitions that may reach a use), uses can refer directly to the unique definition.

As an example, here is an example that is not in SSA form, because `a` is assigned twice:

```
a = 1
b = 1;
a = 2;
return (a + b) * (a + b);
```

A human can easily tell, that the first store to `a` is dead, since `a` is reassigned. It is also easy to tell that the both of the expressions `a * b` compute the same result, since they refer to the same `a` and `b`. However, this is not as easy to tell for a compiler. “Versioning” each definition of a variable and tracking which version gets used makes the observation simpler for an optimizing compiler:

```
a_1 = 1
b_1 = 1;
a_2 = 2;
return (a_2 + b_1) * (a_2 + b_1);
```

Here definition $\mathtt{a}_1$ has no uses, this can be seen by keeping *def-use chains* (which link together all uses of a definition, and which are not to be confused with *use-def chains* mentioned above). But, the fact that $\mathtt{a}_2$ is defined, doesn't make $\mathtt{a}_1$ unavailable and if it held more interesting value then a constant, an optimizing compiler could use it even after the definition of $\mathtt{a}_2$. It is also much easier to tell that the `a + b` expressions are indeed the same, since now they are the result of applying the same operator to the exact same versions of variables, regardless of how many definitions these variables had. The implementation of SSA construction by versioning the definitions is also easy to implement.

Constructing SSA becomes more problematic with conditional execution:

[1] https://llvm.org/
[2] https://gcc.gnu.org/

```
b = 1;
if (a) {
    b = -b;
}
return b;
```

When we try to version the variables here, we don't know whether to continue with $\texttt{b}_1$ or $\texttt{b}_2$ after the `if` statement, though we are able to tell just fine that the use of `b` in the conditional branch corresponds to $\texttt{b}_1$:

```
b$_1$ = 1;
if (a$_1$) {
    b$_2$ = -b$_1$;
}
return b$_?$;
```

The problem is that multiple definitions reach a use. With use-def chains this could have been easily represented. But with SSA we don't want to use def-use chains, since we wish only one definition to reach each use. This brings us to the idea of actually introducing a definition after the `if` statement, which merges $\texttt{b}_1$ and $\texttt{b}_2$ into $\texttt{b}_3$, and can then be unambiguously used by the return statement:

```
b$_1$ = 1;
if (a$_1$) {
    b$_2$ = -b$_1$;
}
b$_3$ = $\phi$(b$_1$, b$_2$);
return b$_3$;
```

Of course, the ambiguity is still there, just hidden behind the mysterious ϕ-function (*phi* function) which provides the merging definition. We would like the ϕ-function to evaluate to right version depending on the control flow which occurs at run-time, so we define the ϕ-function to do exactly that. Though the semantics may seem a bit weird, having the ambiguity hidden in the ϕ-function is is still better than use-def chains, since there can be other uses of $\texttt{b}_3$ all referring to the single ϕ-function, instead of each having multiple reaching definitions.

The simplest possible method for SSA construction inserts a ϕ instruction for every variable at each merge point. This is correct, but such approach introduces many redundant ϕ-functions, that don't do nothing much useful. For example in the example above a ϕ instruction would be introduced after the conditional branch for `a`, even though there is a single definition:

```
b$_1$ = 1;
if (a$_1$) {
    b$_2$ = -b$_1$;
}
a$_2$ = $\phi$(a$_1$, a$_1$);
b$_3$ = $\phi$(b$_1$, b$_2$);
return b$_3$;
```

Similarly redundant ϕ functions are created even in loops:

```
a = 1;
loop:
if (f()) goto loop;
```

Here, even though there is not reference to `a` in the loop, a ϕ function needed to be introduced, since execution may flow to the beginning of the loop from two places (the code preceding it and the end of the loop):

```
a_1 = 1;
loop:
a_2 = ϕ(a_1, a_2);
if (f()) goto loop;
a_3 = ϕ(a_2, a_2);
```

The ϕ refers to itself and only one other value (a_1), so it also can be safely removed and the value used directly instead. Similar “cyclic” ϕ-functions can occur even indirectly—two ϕ-functions can refer to each other and a single other value, which could just be used directly.

So even though even simple SSA construction is possible, the produced code isn't as useful because of many redundant ϕ-functions. The original inventors of SSA form and the ϕ-function concept [Rosen et al., 1988] came up with an efficient algorithm for SSA construction [Cytron et al., 1991], which is based on the new concept of *dominance frontiers*, which in a control flow graph are exactly the places where ϕ-functions are needed. Their algorithm produces what is known as *minimal* SSA form, which doesn't contain redundant ϕ-functions. Despite the name, even more “minimal” forms of SSA exist, for example *pruned* SSA doesn't contain *dead* ϕ-functions.

SSA form is deeply connected to the notion of *dominance* in a control flow graph.

Definition 3.1 Dominance. Let X and Y be nodes in the control flow graph *CFG* of a program. If X appears on every path from Entry to Y, then X *dominates* Y. [Cytron et al., 1991]

3.4.1 Value-based SSA

An important observation with regards to SSA form is, that since each variable is assigned exactly once, and the variables don't get reassigned, there is no need for the concept of a *variable*. The *values* assigned to the variables can be used directly. Values are described by their structure, for example number literals, like `5` or operations applied to other values like `add 1, 2` (addition of two numbers) or `add 5, (sub 6, 7)` (addition of a number and the result of subtraction of of two numbers). Generally we can divide the values used in programs into two categories:

1. *Constants.* These represent number or character literals, but also addresses of static objects. Examples include `3` (integer literal), `'a'` (character literal), `f` (address of a function), `g` (address of a global variable).
2. *Operations.* These represent values produced from other values by applying some arithmetic or other operation. Examples include `neg 5` (unary negation operation applied to a constant), `add 5, neg neg 3` (binary operation applied to a constant and a negation applied to negation of a constant number 3). Arity isn't limited, and for example function calls can be seen as operations on several values of which the first is the called function and the rest are the arguments, e.g. we can have function `f` called with 4 arguments: `call f, 1, 2, 3, 4`.

Function parameters are constants as well—even though arguments (actual parameters) passed to a function may just be results of some operations, from the point of view of the function the formal parameters are constant values that materialize when the function begins its execution.

When represented in a compiler, values are objects and they refer to each other by means of references (for example through pointers). Our textual notation for operations so far was direct: a use of value meant writing out the textual representation of the definition (constructor) of a value. Because each value can be used many times and due to the recursive nature (operations are able to refer to other operations), this quickly becomes unwieldy. We will use a different notation that assigns each value a unique number and for value number i writes the references to it as $\texttt{v}_\texttt{i}$. The definitions are accompanied by an "assignment" whose purpose is to show what index is used to represent references to a particular value. For example, we could have:

```
v₁ = 3
v₂ = 3
v₃ = mul v₁, v₂
```

which differs a bit from:

```
v₁ = 3
v₃ = mul v₁, v₁
```

This notation also easily supports operations that refer to themselves, like the redundant ϕ-functions we have seen for loops:

```
...
v₂ = phi(v₁, v₂)
...
```

It should be noted that our notation for values doesn't imply anything about the order of operations—the only theoretically imposed order is the dependencies among the operations themselves. The references of operations to each other actually form a *direct acyclic graph* (DAG) showing the data dependencies—*data-flow*.

3.4.2 SSA for machine code

Though the *value-based* representation of SSA may be useful in some contexts, it is not always possible to use it. After all, the resources for which we may want to construct SSA form may not represent values at all, or the values may be constrained like shown below. In such cases transforming the intermediate representation to SSA form is still possible, but has to be based on versions and we will call it version-based SSA.

One such example is SSA form for *machine code* (or assembly). Here, we work with registers, and registers no longer *represent* values, they *hold* values. Say we want to represent the following x86-64 instructions in SSA form:

```
mov rcx, rax
add rcx, rbx
```

This example represents the addition `rcx = rax + rbx`, where all three registers are preserved. Obviously, in this example `rcx` is assigned twice. We want to apply the standard SSA versioning construction. The two address encoding actually means that `rcx` is in fact both read and written, but the read `rcx` is different than the written `rcx`. Thus we also need to extend our representation of our instructions to separate the register reads (*uses*) from writes (*definitions*):

```
mov rcx₁ | rax₁
add rcx₂ | rcx₁, rbx₁
```

Above we follow the convention where all definitions are written before the "|" symbol, and all uses after. This way the representation became more similar to classic three address code, where operations are non-destructive.

Other snippets of x86-64 code can also produce a bit surprising results:

$\texttt{cmove rdx, rsi} \quad \Rightarrow \quad \texttt{cmove rdx}_2 \mid \texttt{rdx}_1\texttt{, rsi}_1$

$\texttt{setnz al} \quad \Rightarrow \quad \texttt{setnz rax}_2 \mid \texttt{rax}_1$

$\texttt{xor rcx, rcx} \quad \Rightarrow \quad \texttt{xor rcx}_1$

In the first example we have the conditional move if equal instruction. It moves `rsi` to `rdx` only if the zero bit is set in the flags register. For our purposes the final value of `rdx` can be `rdx` or `rsi` and with no better information, we must pessimistically expect both. So we have two uses and one definition.

Second example has a set byte if not zero instruction, which writes 1 to the lowest 8 bits of `rax` register if the zero bit is not set in the flags register, or 0 otherwise. Since only the lowest bits are set, the instruction actually merges the new low 8 bits with the rest of the `rax` register as set previously, and this has to be modelled through a use of the previous version of the register.

In the last example, the idiom for zeroing out a register is used. Xoring a register with itself produces zero, regardless of the previous value of the register. Compared to move of 4-byte immediate zero, this is much shorter, so the idiom is very common. But, since the register is overwritten with zero, there is actually no reason for the instruction to *use* the former value of the register— the previous value doesn't matter. This is why we want to model the instruction as only *defining* the register. In practice, the processor also doesn't have to wait for the used register, and for example Intel considers using `xor` to zero out a register a *dependency-breaking-idiom* [Intel Optimization Manual, 2023].

SSA on machine code accomplishes its goal—there is only one assignment to each *register version*. But the versions of one register are still tied together. When the SSA form is deconstructed back into machine code, only the version numbers are dropped. So while machine SSA form helps with *analysis*, because tracking definitions became easier, in reality there are still multiple assignments to each resource.

3.5 SSA deconstruction

No common processor provides ϕ-functions on which the SSA form relies. Thus replacement of ϕ-functions with other instructions having same effect is needed. This is step is called *SSA deconstruction*. Since ϕ-functions are the means of achieving single static assignment, necessarily eliminating them means that the program will no longer be in SSA form.

ϕ-function's semantics say that it evaluates to whichever value is appropriate to the control flow happening at runtime. In a representation based on control flow graphs, this is made explicit by basic blocks. We can achieve the effect of a ϕ-function with multiple assignments through a copy instruction in each of the predecessor blocks, i.e. from:

```
block0:
    v0 = 5
    jmp block2

block1:
    v1 = 3
    jmp block2
```

```
block2: block0, block1
    v2 = phi v0, v1
    jmp block2
```

to:

```
block0:
    v0 = 5
    v2 = v0
    jmp block2

block1:
    v1 = 3
    v2 = v1
    jmp block2

block2: block0, block1
    jmp block2
```

This is the original way of eliminating ϕ functions and was introduced by [Cytron et al., 1991]. Though seemingly simple, there are multiple subtle problems, which have demanded improvements in this area.

One of the problems is, that there may be no predecessor, that is suitable for the copy. This can be seen on the following program in SSA form:

```
block0:
    v1 = 3
    branch ⟨...⟩, block2, block4

block2: block0
    v2 = 4
    jump block4

block4: block0, block2
    v3 = phi v1, v2
    ret v3
```

Inserting copies in to predecessor blocks would look like this:

```
block0:
    v1 = 3
    v3 = v1
    branch ⟨...⟩, block2, block4

block2: block0
    v2 = 4
    v3 = v2
    jump block4

block4: block0, block2
    ret v3
```

But the problem is that copy inserted into block 0 for deconstructing the ϕ-function `v3` in block 4 takes effect even in case the execution doesn't actually go to block 4, but goes to block 2 instead. The problem stems from the fact, that control flow can transfer from block with *multiple successors* (where execution continues in only one of the successors) to a block with *multiple predecessors* (where ϕ functions may be needed to merge multiple reaching definitions). In a control flow graph where possible transfers of control flow across blocks are embodied by edges, this is known as a *critical edge*. It is possible to split a critical edge by replacing it with an intermediate block. The block will have only a single predecessor (the block with multiple successors, block 0 in our example) and a single successor (the block with multiple predecessors, block 4 in our example):

```
block0:
    v0 = ⟨...⟩
    v1 = 3
    branch v0, block2, block5
block5:
    v3 = v1
    jmp block4
block2: block0
    v2 = 4
    v3 = v2
    jump block4
block4: block0, block2
    ret v3
```

Now the copies could be inserted into the new basic block 5 and are executed only when control flow actually goes from block 0 to block 4 (through block 5).

Another problem with eliminating ϕ-functions correctly is that semantically it is as if all ϕ-functions in a block executed in parallel at the time the control flow enters the block. This is due to the fact, that ϕ-functions evaluate to the right value based on control flow alone. But copy instructions that we use for replacing ϕ-functions execute sequentially. With multiple ϕ-functions in a basic block, a naive insertion of copies into predecessor can lead to *incorrect translation*.

Briggs [Briggs et al., 1998] identified two examples where this happens, the first is known as the *lost-copy problem* and the swap problem.

Both problems are ultimately due to dependencies among the ϕ-functions and their arguments. Bad order of copies can overwrite needed values. Simplest possible solution is to do the copies in two steps: copy all phi node arguments into temporaries and only then copy from temporaries into the right destinations. The approach is correct, since the first step is non-destructive (unlike the naive copy insertion) and in the second (destructive) step only new temporaries are read, which are not involved in any-ϕ functions. While correct, the approach produces a large number of copies, most of which are not needed.

Briggs [Briggs et al., 1998] approaches the problem as a scheduling problem. Since copy instructions have a destination (definition) and a source (use), he finds an order such that all uses of a virtual register precede its redefinition. If this is not possible due to cycles (like in the *swap* problem), then a temporary and an additional copy needs to be inserted to break the cycle.

Briggs also makes an important observation—the problems only manifest after some of the more aggressive operations are run (like copy folding), and not after others (like copy propagation or dead code elimination). Sreedhar [Sreedhar et al., 1999] calls the form of SSA constructed by [Cytron et al., 1991] and other algorithms *conventional SSA* and notes that it has the important property that all virtual registers in the same *phi congruence class* (containing virtual registers connected via ϕ-instruction) can be replaced by one representative and the ϕ-instruction eliminated.

Importantly, in [Sreedhar et al., 1999] they also consider *transformed SSA*, which doesn't have the *phi congruence property*, and present three methods for transforming transformed SSA into conventional SSA. The first method is remarkably simple: in addition to the copies in predecessor blocks, a copy of the ϕ-instruction is added. But these are different copies than in previous algorithms! Here we are still in SSA form—the copies can actually be considered *aliases* for the original values, or products of an *identity operation* and they obey the single assignment property.[1]

The next two methods described by Sreedhar gradually build on top the first one and additionally make use of liveness (described later in section 3.8.3.1) and interferences (also described in a later section, 3.8.3.2) to prevent inserting too many unnecessary copies.

Full SSA deconstruction based on Sreedhar's approach thus consists of transformation to conventional SSA form, merge of virtual registers in the same phi congruence class ("drop of versions of variables") and simple deletion of the ϕ-instructions.

The most comprehensive treatment of SSA deconstruction was conducted by Boissinot [Boissinot et al., 2009]. Their approach highlights more subtle issues with SSA deconstruction, and presents an algorithm partly based on [Sreedhar et al., 1999], which mainly aims for correctness, but also quality (of the generated code) and efficiency (of the SSA deconstruction). The signature of their algorithm are three passes, which strictly separate between correctness and optimization. Their approach also makes use special properties of SSA to improve the algorithm's run time.

Copies introduced by SSA deconstruction, but also any copies in general, can be eliminated by *coalescing*. Intuitively, if we merge the source and destination of a copy instruction into a single virtual register, then no copy instruction is needed (since it would be a copy to itself). Basic approaches to SSA deconstruction insert copies freely and depend on elimination of the copies through coalescing. But doing some coalescing in SSA deconstruction stage can help greatly, since many fewer virtual registers need to be considered in register allocation stage (speeding up all aspects of it). Also, while in SSA form, many analysis are faster and cheaper to compute and more information may be available to guide the coalescing. Because of these reasons both [Sreedhar et al., 1999] (methods 2 and 3) and [Boissinot et al., 2009] do coalescing as part of SSA deconstruction.

3.6 Instruction selection

Instruction selection is the process which translates a compiler's intermediate representation to machine instructions. While compiler middle end intermediate representations (*IRs*) are usually machine independent and made to be easily optimizable, machine instructions are usually designed to be conveniently executable by a processor. This, irregularities in instruction sets as well as special constraints imposed by individual instructions make the problem of choosing the "best" instructions hard.

To ease translation and split concerns, instruction selection is usually allowed to assume that there is an infinite amount of registers available. These *virtual registers* are then mapped to physical machine registers in a later phase, called *register allocation*. We describe register allocation in a later section (section 3.8), where we also discuss the problems of *machine instruction constraints*, where some instructions only work with some physical registers (section 3.8.3.6). In this chapter we will use the prefix `t` and a number (for example `t12`) for virtual registers and assume that there is unlimited number of them.

[1] In fact, exactly because these SSA values are aliases, they are folded by algorithms like *copy folding*, that result in transformed SSA form, which exhibits problems with naive SSA deconstruction.

If the middle end IR and machine instruction set are relatively close, then even something as simple as one-to-one mapping is possible. For example, this middle end IR:

```
v3 = add v1, v2
v4 = load v3
```

can be translated to the following RISC-V code:

```
add t3, t1, t2
ld t4, 0(t3)
```

The mapping is relatively easy, because with RISC machine instructions, all operations are done on *registers*, and only load and store instructions touch memory. This closely resembles the three address code used by the IR above. Even on CISC architectures (which are known for addressing modes allowing memory operands), translation of RISC-like middle end IRs is still straightforward, since simple forms of instructions are generally still available and the more advanced addressing modes will simply not be used. For example, this is how the piece of IR above could be translated to x86-64:

```
mov t3, t1
add t3, t2
mov t4, [t3]
```

The first copy instruction is needed to compile the IR's three address code to x86-64 two address code. Use of advanced addressing modes and memory operands in arithmetic instructions can improve the generated code considerably:

```
mov t4, [t1+t2]
```

Even though we can intuitively tell that the second x86-64 version is "better", it is very important to consider exactly why. *Fewer instructions* are used, but is the code *smaller*? Does it execute *faster*? Does "faster" consider throughput as well as latency? These questions should be considered by instruction selection and their answers should guide the instruction selector to find better code. Often, instruction selectors operate in a *cost* based model, where each instruction is given a cost and, if possible, instructions with the lowest cost are chosen. This cost should ideally consider all mentioned factors, but often it can be hard to balance between speed and code size trade-off, and since both can be important in different scenarios, having different instruction costs for different situations can make sense.

Different orders of instructions can have impact on execution speed as well. Usually, this problem is not considered as part of instruction selection, and separate *instruction scheduling* pass is used to reorder instructions to a more favorable order. Instruction scheduling is described in 3.7.

3.6.1 Outline

The first two instruction selections presented above, can be done relatively simply programmatically. Each IR instruction is in turn *expanded* into one or more machine instructions. Registers (or more precisely *virtual registers* in this stage before register allocation) are used as means of abstraction—each instruction reads operands from registers and writes results to registers. Different operand kinds (such as immediates or memory locations) are handled by special few instructions that are able to load immediate or perform load from memory or store to memory. The use of registers has to be consistent, e.g. if an instruction maps to multiple instructions which need a temporary register, none of the input registers should be used, since later IR instructions may expect the registers to not be modified. Either a fresh virtual register has to be allocated, or one of the destination registers can be used, since they will be overwritten by the expansion in any case.

This method produces a simple one-to-many translation. While the instruction sequence emitted for each single opcode can be optimal, most of the interesting instruction selection opportunities stem from the fact that there are instructions which combine several operations. In same cases this means many-to-many or even many-to-one mappings.

There are three fundamentally different approaches for handling instruction selection. Most existing algorithms for instruction selection fall into one of these three categories, though even algorithms falling into the same category can differ significantly in their speed (how fast they run), quality (how fast does the produced code run) and capabilities (e.g. ability to handle instructions with multiple outputs, like `idiv` on x86-64). In the following subsections we introduce the three approaches.

3.6.2 Peephole optimization

Peephole optimization is a general optimization technique (see section 3.3). It is applicable to any compiler stage, but in the back end peephole optimization is often associated to final improvements to machine code after instruction selection, instruction scheduling and register allocation. But peephole optimization has been applied even to perform instruction selection. To distinguish this, we will call the technique *instruction selection by peephole optimization.*

The idea for instruction selection by peephole optimization is simple. The naive expansion method produces a simple one-to-many translation, and we can improve it, by a peephole optimization pass that can find patterns and replace multiple instructions with either shorter or more efficient instruction sequences. Since a lot of expansions are mere translations one-to-one, even a relatively small peephole window can actually examine many instructions in the original IR.

But, while the peephole optimization can be done on the machine instructions themselves, they are not much suitable for optimization. Apart from that, they are also machine specific, and while improvements to instruction selection are machine specific by nature, it is preferable when a single mechanism can drive the optimization for many different instruction sets. Also, writing the peephole optimization patterns by hand is tedious and does not scale for a compiler targeting many architectures.

Influential in this are was the algorithm by Davidson and Fraser [Davidson et al., 1980]. Their retargetable peephole optimizer is based on *register transfers*, which describe very low level machine operations—like actual transfers among registers, but also arithmetic operations and settings of flags. Each machine instruction has an associated *register transfer list*, which describe the effects of the instruction. Their algorithm then operates in four passes:

1. Translate all machine instructions to register transfer lists.
2. Iterate over the register transfer list backwards and determine the observable effects (register transfers) of each list.
3. Iterate over the register transfer lists forwards and check for each pair of whether there is an instruction whose register transfer list has the combined effects of the pair. If there is, combine the two register transfer lists into one.
4. Iterate forward over the code and for each register transfer list find an instruction that implements it and emit it.

The first step is actually optional, since the compiler can generate the register transfers directly, but nominally their algorithm both reads and writes machine instructions. The register transfers are only an immediate step, and importantly the backwards pass deletes unobservable effects, which don't need to be implemented by the combined instructions. The effects are mainly assignments to registers and flags, which Davidson and Fraser handle in a unified way. Since an assigned (virtual) register or a flag may not ever be read before being assigned again, it may be deleted. Liveness or rather death of registers is also important for deletions: if a definition gets "inlined" into all its uses (e.g. a small constant on an architecture which has addressing mode allowing small immediates), then the definition can be deleted only if it is determined to no longer be live after the deletion (*becomes dead*). The deletion may take form of either combination into an instruction which doesn't have the extraneous effect (in phase 3) or replacement by a single final instruction (phase 4) which also doesn't need to have the extra effect. This is a form of dead code elimination (since for example an empty register transfer list doesn't translate to any instruction). But the extraneous assignments are kept if there *isn't* an instruction that doesn't have them (for example, most arithmetic instructions set flags, even if they are not needed, and there may be no equivalent instruction that doesn't set the flags).

Important aspect of Davidson's and Fraser's approach is that the correspondence between register transfers and machine instruction is described by a textual file called *machine description*. The compiler interprets the file and uses it to map instructions to register transfers (phase 1), replace pairs of register transfer lists (phase 3) and to map register transfer lists back to instructions (phase 4). All of these phases use the textual description and are thus based on operations with strings. This has the advantage that the compiler is easily retargetable just by swapping out the machine descriptions. Since in the last phase register transfer lists are mapped back to instructions, the algorithm has to obey an invariant: any produced register transfer list is implementable by at least one instruction. This is why phase 3 is guided by the machine description and why the machine description has to be written with care. In phase 4, if more machine instructions can implement the same register transfer list the first one in the machine description is chosen (simply because the algorithm tries to match with all instructions one-by-one). This can be used to model (relative) costs of instructions with equivalent behavior.

Later improvement of the algorithm by the same authors [Davidson et al., 1984a] better formalized the phases of the algorithm. The *expander* produces register transfers, *combiner* combines pairs of instructions and *assigner* translates register transfers back to machine instructions (and in their approach also performs register allocation). Combiner and assigner are now improved and based on finite automaton generated at compile time of the compiler (*compiler-compile time*), making them much faster. Though the simulation of combined effects of instructions happens still on strings. *Cacher* is a new addition and performs local common subexpression elimination by keeping track of what values are available in what registers in a basic block. Another powerful addition was consideration of not only *physically adjacent* instructions, but also *logically adjacent* instructions—instead of considering instructions in their *control flow* order, they are considered based on *data flow*. This allowed to consider simple definitions (like constants) to be folded into instructions, even if they fell out of the small peephole window.

Davidson's and Fraser's design turned out to be very influential in the area of peephole optimization. In particular the idea of expanding the instructions into a machine independent representation, that exposes even the smallest operations, then performing optimizations and then combining the operations into machine instructions. This aspect is kept by a lot of compilers using peephole optimization for instruction selection. The idea of a machine description that is preprocessed at compiler-compile time also stuck. Slow compiler compile time doesn't matter as much as the speed of the compiler itself.

Other aspects of the original Davidson-Fraser design didn't catch much. Later algorithms opted for a *fixed sets of patterns*, in place of deriving them on the fly with symbolic execution. These patterns used to be written by hand, but considering the existence of the machine descriptions and symbolic executors based on them, Davidson and Fraser [Davidson et al., 1984b] used them to derive the patterns. They did this by introducing a set of programs known as the *training set*, over which they run their classical peephole optimizer [Davidson et al., 1984a]. Each unique optimization performed on the training set is written to a file, which is then used as base of a more limited peephole optimizer, which operates only based on these patterns. While the files are still text based, they use *string interning* based on a hash based data structure to avoid comparing strings and to speed up string comparisons.

3.6.3 Covering

A different paradigm for approaching instruction selection is *covering*. With intermediate representations like trees and DAGs (e.g. with value-based SSA on a control flow graph from section 3.4.1) the operations and their operands form graphs—in particular *data-flow graphs*, because the graphs capture the flow of values from their definitions to their uses.

Also instructions can be represented as graphs, or rather graph-based *patterns*. For example an `add` instruction, which accepts two registers and produces a result in a register, can be represented as a node (the result of the addition) with two children nodes (the operands). A different pattern can represent a similar instruction, but with another addressing mode, e.g. an `add` of a register and an immediate or `add` of register to memory location.

Covering instruction selection covers the data-flow graph of the middle end IR with the instruction patterns. As each instruction pattern is associated with an instruction (or instruction sequence), the covering gives the final set of instructions, but not necessarily their exact ordering—any order resulting from a bottom-up walk is a possible final instruction sequence.

As nodes are gradually covered, results from operations themselves become operands covered by instruction patterns and the produced. The nodes in the instruction patterns are typed by the kinds of operands of the x86-64 architecture. For the x86-64 architecture for example, at least the following fundamental types are required:

- Registers.
- Memory locations
- Small immediates (32-bits)
- Integer constants (64-bits)

Small immediates can be used anywhere where full integer constants are allowed. A "retyping" pattern is usally associated with these transitions. There is no actual instruction associated with such pattern. Leaves in patterns are special and may need special separate types to be represented. For example, access to stack allocated variables requires access to the base pointer which is not produced by anything. A "retyping" no-op pattern, which makes base pointer (leaf) available as register will then allow the frame pointer to be usable like registers produced by other patterns, for instance loads of constants, arguments or results of arithmetic operations.

The formulation on *graphs* is very general and allows both the IR which is pattern matched and the patterns to be general graphs. In practice, simpler forms sometimes suffice and we stick to three categories as described in [Blindell, 2016]:

1. *Trees.* Trees are natural when the IR is an AST or a low level version of it—i.e. only with simple RISC-like close-to-machine operations, which are more amenable to matching. Many instruction patterns also form trees—most arithmetic operations produce one result from multiple operands.
2. *DAGs.* While trees are nice and simple, they are often too limiting in an optimizing compiler. For example, when common subexpressions are eliminated, a single node can be used by multiple operations, but this means that the node no longer has a single parent, but multiple. Directed acyclic graphs (DAGs) are able to represent this exactly.

   DAGs are sometimes also necessary for exact modelling of instructions, which don't correspond to trees. Notable example are *multi-output* instructions which produce from some number of inputs multiple results. An example is the x86-64 instruction `idiv` which produces two results—the quotient and the remainder. We may even consider settings of *flags* to constitute additional outputs of instruction—in that case on the x86-64 most arithmetic instructions would be multi-output.
3. *General graphs.* Both trees and DAGs are unable to capture control flow, which in non-trivial programs contains loops and hence *cycles* in the control flow graphs. Most advanced instruction selection for an entire procedure (across basic blocks) thus requires instruction selection to be performed on general graphs with suitable patterns that are able to cover such graphs.

Importantly for multi-target compilers, even though the patterns themselves are machine specific, their form is machine independent. A single algorithm can work for any architecture, as long as it is provided a suitable set of patterns and instructions that correspond to them.

Wide variety of algorithms have been used to perform the covering. They range in what graphs they are able to handle, quality, run time and complexity. In the following subsections we introduce some of the existing approaches.

3.6.3.1 LR parsing

The algorithm of Glanville and Graham [Glanville et al., 1978] uses LR parsing for instruction selection.

LR parsing [Knuth, 1965] is a technique that is usually used for transforming streams of tokens into parse trees or ASTs. However, it can be seen as a general technique that allows matching of tree patterns. The productions in a context free grammar can be seen as the patterns, where non-terminal symbols are the roots of patterns and correspond to the types of nodes. Terminal symbols are then the trivial constants like the base pointer or concrete integer literals—leaves.

The advantage of LR parsing is that it is well studied. The parsers can be generated from context free grammars based on the instruction patterns. The expensive preprocessing is done at compiler-compile time and the generated parsers themselves are fast. Additional predicates (e.g. checks of small constants) and actions can be added to the LR parsing through the use of *attributed grammars* [Knuth, 1968].

A fundamental problem with LR parsing in the area of instruction selection is, that it expects a linear sequence of terminals—the tree structure has to be linearized and the matching happens over that linearization. Additionally, the patterns for instruction selection are usually highly ambiguous—many instructions can produce the same effects. A method for resolving the numerous reduce-reduce and shift-reduce conflicts is needed. Often shifts are preferred to reductions, because they are likely to result in bigger matches. Some kind of cost model is useful for resolving reduce-reduce conflicts.

Other problematic areas include ensuring of correctness (where we want to guarantee that for any possible stack we can perform a reduction) and the size of the generated parser, which reportedly can get impractical [Aho et al., 2006].

3.6.3.2 Top-down matching

LR parsing, like other algorithms which we will consider next, works bottom-up. I.e. matching starts from the leaves and gradually covers nodes with patterns until it reaches the root. An approach working top-down is possible, and natural for recursive formulation like matching on trees.

Cattel [Cattel, 1978] mentions an algorithm which he calls “maximum munch”, which top-to-bottom does the following for each node:

- Try matching all patterns with the current node as root. Select the first matching.
- Apply the algorithm recursively for each child implied by the pattern.
- Emit code associated with the selected pattern.

This is a greedy algorithm. The algorithm prioritizes patterns through their ordering. In Cattel’s algorithm *large* patterns are ordered first, but attention is also paid to their cost per covered node. Larger patterns are usually better, because they implement more of the tree with fewer instructions. Though doing these decisions top down misses a lot of low level context found only at the bottom.

The biggest advantage of maximal munch is its simplicity. But the limitation to trees, and the worse quality (due to greediness) make it not as practical.

3.6.3.3 Bottom-up rewriting

LR parsing is too problematic and top-down too weak due to lack of any backtracking. However, there are better approaches for bottom-up instruction selection. A group of algorithms is based on *dynamic programming*. One such algorithm is [Aho et al., 1989], which operates in three passes:

1. In a top-down fashion for all nodes the applicable patterns are identified. (The top-down pattern matching is interestingly done with an extension of the Aho-Corasick string matching algorithm [Aho et al., 1975].)
2. Bottom-up, the costs for each applicable pattern at each node are computed. All possible types of the node must be considered, since in the bottom-up approach we have not yet determined the type. Thanks to dynamic programming, the costs of children (as all applicable types) are available when costs of parents (for all applicable types) are computed. For each type only the cheapest pattern and its cost is saved.
3. Top-down, the cheapest type is selected for each node. Based on the rule associated with type, the types of children are determined, which allow its cheapest type to be selected.

Only the cheapest patterns for each type are interesting, because while any mode can ultimately be chosen for a node in the third pass, we will only need the cheapest pattern of a certain type. But the type itself is not known until the third pass.

The conflicts that hindered the LR parsing approach are gone—the results either identify the cheapest tiling uniquely through the type of the root node, or it can be chosen arbitrarily if multiple types have the cheapest cost.

A tool called `twig` is described as part of [Aho et al., 1989]. It takes a textual grammar of the tree patterns and generates a C program that does the instruction selection.

The concept of BURS (Bottom-Up Rewrite Systems) was introduced as means of tree covering for instruction selection by [Pelegrí-Llopart et al., 1988]. With BURS it is possible to move the dynamic programming to compiler-compile time. The technique is based purely on rewrites at the bottom of the trees and something like the first pass of [Aho et al., 1989] is not needed.

A tool called `burg` [Fraser et al., 1992b] is based on the BURS theory and does the dynamic programming at compiler-compile time. This unfortunately means some restrictions on the grammars. A similar tool called `iburg` [Fraser et al., 1992b] remains more flexible by doing the dynamic programming in compile time.

3.6.3.4 DAG covering by tree covering

The algorithms introduced in previous subsections were all limited to trees, i.e. only tree programs and tree patterns were allowed. In general, extending the tree approaches to *DAG programs* is possible by transforming the DAGs into trees. Two fundamental methods are described by [Blindell, 2016] and both work by identifying the *trees with multiple parents* (i.e. the common subexpressions), on which is possible to either:

1. *Split the edge.* The tree is removed from the graph and replaced by a special leaf. All individual trees are then covered separately and the results merged.
2. *Duplicate the node.* The common tree is duplicated for every use.

Both of these approaches are problematic: splitting makes small trees, which can miss covering with bigger patterns (of supposedly better instruction) and duplication undoes so carefully done common subexpression elimination.

A notable application of duplication is Ertl's [Ertl, 1999], which is able to reduce the need for duplication for a class of DAGs. Edge splitting has been successfully used by Koes and Goldstein [Koes et al., 2008].

Unfortunately these straightforward extensions of tree covering don't easily support *DAG patterns*, which are required for multi-output instructions like `idiv`.

3.6.4 Reduction

Trees can be covered in linear time with dynamic programming. Covering of DAGs and (as well as general graphs) is NP-complete [Koes et al., 2008]. Several approaches then instead of solving instruction selection directly, choose to reduce it to another NP-complete problem, use an existing efficient solver, and then transform the solution to the selected instructions.

As solvers of some NP-complete problems have gotten much bigger attention, this approach can be very efficient in practice. For example, techniques based on integer programming [Wilson et al., 1994] and partitioned boolean quadratic problem [Eckstein et al., 2003] exist.

The advantage is, that by embracing the NP-completeness a lot of flexibility can be gained, and the techniques can work across basic blocks, selecting instructions for whole functions.

3.6.5 Practical considerations

Using register transfers for representing instructions has an interesting parallel in the processor itself. Modern x86-64 implementations like the ones from Intel [Intel Software Developer's Manual, 2023] split instructions into *micro-operations*. These micro-operations are small units of work that the processor actually executes. Because the processor doesn't execute instructions, but only micro-operations, the instructions can be seen just a compact or even *compressed* representation of micro-operations lists. Register transfers can be viewed similarly as the micro-operations. Combining multiple register transfers and register transfer lists can then be seen as *compression* producing the fewest instructions having the same effect.

Processors sometimes also find it more valuable to keep some operations together. This combination of multiple instructions or micro-operations into a single micro-operation are called macro-fusion and micro-fusion. They are used for example for comparisons followed by branches.

In a way, we can say that the processor receives a compressed form of instructions and reencodes it to a form suitable for execution.

3.7 Instruction scheduling

Instruction selection selects the machine instructions implementing the desired behavior. However, these instructions are not necessarily ordered well to suite the concrete machine. It is the job of instruction scheduling to decide on the final order of the instructions.

The dependencies among instructions (selected by instruction selection) impose a *partial order* on the instructions. These dependencies include data-flow on registers (i.e. operands have to be evaluated before the operations that use them), but also memory operations. Loads and stores to memory can be very limiting to the ordering, though *alias analysis* may be able to tell which stores and loads are safe to reorder.

Instruction scheduling then produces a *linear order* of the instructions, where the dependencies are honored, but the order is more suitable to to make better use of machine resources.

3.7.1 Motivation

In a processor, that processes and executes exactly one instruction per cycle, there is not much needed to find better orders of instructions—when all operations take the same time, any ordering will produce the same net result.

In pipelined processors this becomes more interesting, because although the processor still executes one instruction per cycle, *multiple instructions* are processed at the same time in the pipeline. Each cycle a new instruction enters the pipeline and one finishes its execution. Hence there are as many instructions in the pipeline as there are pipeline stages and progress is made on each instruction in each stage of the pipeline. This is comparable to to an assembly line in a factory. In a usual pipelined processor we can imagine the following stages:

1. *Instruction fetch.* The instruction is loaded from memory.
2. *Instruction decode and operand fetch.* The processor figures out the nature of the instruction and loads the operands.
3. *Execute.* The instruction is executed.
4. *Write-back.* The results of the instruction are written back to registers or memory.

In this scheme, the processor must be careful of the dependencies, since if the results are written in stage 4, but operands loaded in stage 2, then a dependency in consecutive instructions makes them unable to be executed in this pipeline. E.g. in:

```
add rax, rbx
add rax, rcx
```

The first instruction writes `rax` in its stage 4, but the second instruction reads `rax` in the first instruction's stage 3 when the results are not yet available. The processor has to detect this, and often has to *stall* the execution until the results become available. Actually, here the result written back at stage 4 is already available at stage 3 and may be *forwarded* to the stage 2 of the preceding instruction at the same cycle. But in a real pipelined processor there is a higher number of stages, which makes dependencies of adjacent instructions much more problematic.

For pipelined processors, it is beneficial to *interleave* unrelated computations—while one waits for the results, the other can be executed freely. If, like above, we don't have anything to interleave with, we can't improve the code. But, in long basic blocks opportunities for interleaving may occur.

3.7.2 List scheduling

Cooper and Torczon [Cooper et al., 2004] describe a technique for scheduling instructions in a basic block, that is the base of many other, more advanced techniques. The approach works in four stages:

- *Rename.* Registers are renamed to prevent false dependencies.
- *Build dependency graph.* In a backwards fashion the operations in the block are examined, and edges from definitions to uses are added.
- *Assign priorities.* Priorities are assigned to be used to later to guide the scheduler.
- *Schedule operations.* A *ready list* containing the instructions that can execute in the current cycle is maintained. Instructions are removed from this list, until it becomes empty. At which point (based on the cycle count and prepared operands), new instructions can become ready.

3.7.3 Practical considerations

Even though in the previous subsections we motivated and presented a suitable approach for scheduling, these days the assumptions we made about pipelined processors are mostly obsolete.

No longer, are processors just pipelined. Modern x86-64 processors are much more advanced and feature [Intel Optimization Manual, 2023, Intel Software Developer's Manual, 2023]:

- *Register renaming.* Instead of using only the physical registers, there are many more *architectural* registers and for each result a new fresh architectural register is allocated. This removes false dependencies.
- Out of order execution. The processor doesn't process instructions one by one. Although it decodes them in a sequential fashion, they get processed separately and independently, and only *retired* in their nominal order in a reorder buffer. This is supported by:
  - Multiple *execution units.* Instead of being able to execute only one operation at a time, multiple operations can be executed in different execution units, though each supports only certain kinds of operations. Many operations can be executed in parallel.

  - Issuing of multiple instruction in a single cycle. This way, multiple execution units are kept saturated, because many operations can start their execution at once.
- *Branch prediction* and *speculative execution.* The processor is able to continue executing code after a branch even before evaluating the condition. If the assumption turns out incorrect, it is able to backtrack.

Ultimately, this means that the processor redoes all the instruction scheduling by itself *dynamically* at run time. This is very powerful, because with the addition of branch prediction and speculative execution, it works across basic blocks and adjusts to the actual execution.

This makes *static* instruction scheduling in a compiler a less important optimization. It is still useful, but becomes interesting with at least some of the following:

- expensive operations (like division, which can only execute in a small number of execution units and has high latency),
- precise processor specific information (about execution units, reorder buffer size, instruction latencies and throughput, etc.),
- long or computation heavy basic blocks (i.e. the static scheduling has something to work with),
- predictable branches (i.e. the compiler statically knows which blocks are likely to execute after each other).

3.8 Register allocation

Register allocation is the last of the three big conceptual parts of a usual compiler back end. Motivation, importance and possible approaches are introduced.

The x86-64 architecture is what we mainly care about in this thesis. Since it is familiar, we will be using it as examples in the following sections. It is also a good candidate because it brings some challenges not found on other architectures, but shows the general problems just as well as other architectures.

Note that like with instruction selection although we are already working with target specific instructions, their form doesn't necessarily have to be target specific. Target independent representation of target specific instructions allows us to share also register allocation logic for all targets.

3.8.1 Motivation

During previous phases of the compiler we used a powerful abstraction, we pretended that there is an infinite amount of registers. This is very important for the middle end IR, since it is supposed to be platform agnostic and rather than limiting to some fixed number of registers (per architecture or wholesale), we might as well pretend to have infinite amount of them. But once we start translating the middle end IR we just need to limit ourselves to fixed amount of registers somewhere.

After instruction selection (which determines what instructions to use) and instruction scheduling (which refines the order of the instructions), a snippet of input to register allocation can look as follows:

```
mov t1, 1
mov t2, 2
mov t3, t1
add t3, t2
```

There are several things of note here. The instructions don't operate on real machine registers (like `rax`), but on *virtual registers* (often also called "pseudoregisters" or "temporaries"). It is the goal of register allocation to transform the code so that *physical* ("machine") registers are used. Since the whole program doesn't use more than 16 registers, we have no problem assigning x86-64 registers directly, for example in the order of the temporaries:

```
mov rax, 1   // t1 = rax
mov rbx, 2   // t2 = rbx
mov rcx, rax // t3 = rcx
add rcx, rbx
```

Even for such a simple example, we can notice several things about register allocation alone:

- We introduce a third register `rcx` to store the result of addition. This works well and fits into the 16 registers we have available. But we can notice that after the addition we no longer need the value stored in register `rax`. This is on of the big ideas in register allocation, we only need to store those values that will be needed in the future, and we can use that to "reuse" registers.
- If we were to reuse `rax` for storing the result of addition, our situation would look like this:

  ```
  mov rax, 1   // t1 = rax
  mov rbx, 2   // t2 = rbx
  mov rax, rax // t3 = rax
  add rax, rbx
  ```

  Move (copy) of a register to itself is a no-op, the instructions doesn't have any real effect. It doesn't even change flags, to it is safely possible to remove it. We can notice that the two address code generated from SSA three address code can be improved if it turns out that the destination can be the same register as the first source (or the second source in this case, since addition is commutative).

  Though this brings a question to which we will come back later: Can the register allocator remove the instruction? Does it have sufficient information to do so? Or should it even be concerned about the semantics of instructions it is working with?

While we have shown that opportunities for register reuse arise, it doesn't mean we can't get out of registers. After all, there is a limited number of them, and opportunities for reuse come only when a virtual register is no longer needed later. Even relatively simple expressions can produce code which requires surprising amount of registers, while not allowing much reuse. For example, compilation of the expression `1 + (2 + (3 + 4))` can produce code like the one below:

```
mov t1, 1  // t1 = rax
mov t2, 2  // t2 = rbx
mov t3, 3  // t3 = rcx
mov t4, 4  // t4 = rdx

mov t5, t3 // t5 = rcx
add t5, t4

mov t6, t2 // t6 = rbx
add t6, t5

mov t7, t1 // t7 = rax
```

```
add t7, t6
```

Possible register allocation is noted in the example. The right associative nature of the expression means, that for each of the additions while the left hand side (the immediate numbers) are evaluated first, its result has to be kept in registers until the right hand side is evaluated and ready for the addition. Register reuse is possible but only after the additions, because each has at least one argument that is not needed further. The example could be extended to exhaust all available registers. Contrary to that the left associative version (i.e. `((1 + 2) + 3) + 4`) needs just two registers:

```
mov t1, 1  // t1 = rax
mov t2, 2  // t2 = rbx
mov t3, t1 // t3 = rax
add t3, t2

mov t4, 3  // t4 = rbx
mov t5, t3 // t5 = rax
add t5, t4

mov t6, 4  // t6 = rbx
mov t7, t3 // t7 = rax
add t7, t5
```

Since the operands are kept in registers only for short time in between the additions, there are more possibilities for reuse. So even though both versions use the same amount of *virtual* registers, they need different amounts of *physical* registers. While instruction scheduling, or perhaps instruction selection or middle-end optimizations can transform the right associative version to the left associative one, or just fold the computation entirely (since it's a sum of four constants), the example illustrates that virtual register which are needed for a long time are a problem, since they prevent register reuse and that because of that even simple examples can get out of registers.

3.8.2 Spilling

We have to make all values in virtual register available wherever they are needed. But there may be too little of *physical* registers to do so. One possibility of reducing the *register pressure* is to use memory. In the simplified view of a compiler, there is essentially infinite amount of memory available, so storing values does not deplete a limited resource as much as using physical registers does.

Techniques that involve using memory to reduce register pressure are usually called *spilling*—alluding to the fact that what does not fit into physical registers is put somewhere else, and that in fact it is an undesirable thing and we use it only when absolutely necessary.

The best place for storing spilled values is on the system stack (also often called "frame stack" or "call stack") in the function's call frame. This is due to the same reasons why it is a good place for local variables—each invocation of a function gets its own locations for storing the values. This keeps the functions reentrant and for example naturally supports recursion.

Architectures usually also have a dedicated register for the pointer to the top of the stack, which means that code needing to access the values not fitting into registers can address their memory locations using relative addressing with small offsets, also a feature efficiently supported by all common architectures these days. On the other hand accessing static slots of memory would pose similar challenges as accessing global variables does (e.g. relocations, position independence, large constants, ...).

3.8.2.1 Using spilled values

Putting values into memory brings in the problem of using them in instructions. The generated code used operations involving registers and often instructions don't allow memory locations to be used everywhere where registers are allowed to be used. In fact, on processors employing the "load-store" architecture (which is one of the signatures of RISC processors), there are only few instructions for loading values from memory into registers and a few instructions for storing register contents into memory and no other instruction can address memory locations. But, the computation of a value still needs to store it into a register, just like the use of the value needs it to be present in a register—though between the definition and use, the value can reside in memory. To achieve this load and store instructions are introduced. These inserted loads and stores are called *spill code*.

What makes spilling beneficial, is that the registers involved in the spill code (and in the associated definitions and uses of the spilled values) are only used very locally. Additionally, the registers used for storing and loading are essentially completely independent, because each load and store can use a different register. This makes register allocation much easier (or even possible), since this essentially introduces new and much less "constrained" virtual registers.

In this text, we care especially about the x86-64 architecture, which, like most CISC architectures, doesn't have a load-store architecture. There are for example instructions, which can perform arithmetic directly on locations in memory. Though at most one operand of an instruction can be a location in memory, the other one has to be either a register or an immediate value. So on one hand, the problem of having to use registers for storing/retrieving spilled values remains, but on the other hand, since one operand can be a location in memory, we can take advantage of that and not use an intermediate register at all.

Even though the x86-64 architecture allows the instructions to operate on memory operands, an implementation of the architecture in for example some new Intel processors splits these instructions into micro-operations based on the load-store architecture, using internal *architectural* registers (not accessible directly) for storing the intermediate values. Because of this there probably isn't any *direct* performance difference of using the memory operands. But it is still very useful to use these more complex instructions—not only can the instruction encoding be shorter (thus sparing the instruction cache of the processor), but we take advantage of the internal architectural registers, that we normally wouldn't have access to, which can mean less constraints on the use of the ordinary *physical registers* that we have access to, which may allow storing more values into registers instead of memory and hence have a significant *indirect* impact on performance.

For example, suppose that `t3` needs to be spilled in the following:

```
add t3, t2
```

The straightforward solution is to add a load before and store after:

```
mov t4, [rbp+s3] // s3 = an offset to t3's spill stack slot (immediate integer)
add t4, t2
mov [rbp+s3], t4
```

We have to be careful about actually inserting loads and stores, because `t3` is both *used* and *defined* in this instruction—the instruction essentially does `t3 := t3 + t2`. Because of this, the register used for loading `t3` from memory is the same one that will hold the result that needs to be stored back into memory, so the store needs to use the same virtual register the load does, here it as `t4`.

This example shows, that as mentioned, at least with a very narrow local view, and with the first straightforward solution, spilling `t3` doesn't help with the use of registers. We introduced another pseudoregister, `t4`, to substitute `t3`, but the original instruction just became surrounded by memory operations. Indeed, spilling helps only in a broader scope, where for example `t3` had more definitions and uses.

We can alternatively just operate on the memory location:

```
add [rbp+s3], t2
```

This seems beneficial, since the processor will likely do the same three fetch modify write operations we had with with our own spill code, it will use an architectural register to do so, and we don't need any physical register (represented by above by `t4` virtual register) to do so. But we shall look at this in bigger context than a single instruction. The x86-64 two address code is often generated from three address code (likely from SSA form), for that the code generator likely had to introduce a copy to preserve the value of the first operand[1], so in fact assuming that the original IR was:

```
v3 = add v1, v2
```

the full x86-64 code would be:

```
mov t3, t1
add t3, t2
```

Now the prospect of naively spilling `t3` seems even worse:

```
mov [rbp+s3], t1
mov t4, [rbp+s3]
add t4, t2
mov [rbp+s3], t4
```

For the copy instruction, instead of using a temporary, we can just copy to the spill location immediately (see section 3.8.2.2). But with the copy instruction, the code generator hoped that by assigning `t1` and `t3` the same register, the move instruction could be eliminated. Since for some reason `t3` was spilled, that is now out of the question, but there is still a suboptimality—`t1` is copied to memory and then immediately loaded back again into `t4`, because it needs to be used in the `add` instruction and then also stored back into memory. Use of `t3`'s memory location as the first operand of the `add` instruction helps:

```
mov [rbp+s3], t1
add [rbp+s3], t2
```

But the issue is just hidden—the memory operations are still there, the CPU still has to load the value `t3` from memory in the `add` instruction, when we just had it in a register. Just because the addressing modes of x86 allow use of memory locations in the instruction encodings, doesn't mean that internally the ALU (Arithmetical logical unit) can suddenly operate directly on memory, the CPU still has to internally load the value into a register. So while we spare a general purpose register and instead use an architectural one, we still excessively operate on memory.

[1] Technically, the lowering step doesn't have to generate the copy to preserve the value if it is not needed. But, as mentioned further, this requires non-trivial liveness analysis and eliding (coalescing) the copies is what register allocators try to be good at, so emitting the copy unconditionally is reasonable solution.

Alternatively, starting from the straightforward 4 instruction sequence, we can just forward the load and use `t1` to populate `t4` directly:

```
mov [rbp+s3], t1
mov t4, t1
add t4, t2
mov [rbp+s3], t4
```

Now the dead store to `[rbp+s3]` is even more apparent, and can be optimized away:

```
mov t4, t1
add t4, t2
mov [rbp+s3], t4
```

Now, we keep the values in registers the whole time, and only store the final result in memory on assumption that later the value is needed and storing it in memory somehow helps the register allocator.

Another important observation is that, the "optimal" code we ended up with, is very similar to what we started with—there is just a store instruction in the end. Whether spilling `t3` helped is impossible to tell from this context. But we can something about the possible benefits of spilling `t4`—there are none. By spilling `t4` we would get the same instruction we already have, just with a different pseudoregister (say `t5`). We are at the `t3` spilling *fixed point*. For this reason we should prevent the register allocator from thinking that spilling `t4` might be a good idea, since it might lead it to an infinite loop. The reason why spilling `t4` is not beneficial is the fact that the potentially inserted loads and stores are redundant. The more general pattern is, that any definition immediately followed by use is not a possible spill candidate. This includes virtual registers inserted for spill code (so prevents potential infinite loops spilling spill registers), but can also prevent spilling other virtual registers, which otherwise might look like plausible spill targets, but in fact aren't.

A different point of view is, that essentially, in (the last snippet) in the first two instructions `t3` is represented by `t4`, while in the last instruction it shifts back to being represented by `t3`, which due to some previous decision, resides in a memory location `[rbp+s3]`. It is as if we have originally split the register `t3` into multiple registers (`t3` and `t4` connected by moves) and only spilled `t3`. It results in the same great code as our optimized spilled version improved the generated code, because some of the multiple registers can be much less constrained than the original one and are thus less likely to be spilled and more likely to get assigned a physical register.

In contrast, we argued, that by merging `t4` and `t1` we would have had the chance to eliminate the move instruction, thus improving the code as well. Both "merging" (called *coalescing*) and "splitting" (called *live range splitting*) can improve the code in different situations, this makes it even harder, because recklessly doing one or the other will make the code certainly worse, while doing neither may be just as bad. This makes great register allocation even harder.

3.8.2.2 Spilling in copy instructions

In the previous section we found the need to spill a virtual register used in a copy instruction (often called *move*), like this one:

```
mov t1, t2
```

If `t2` is to be spilled, we nominally need to load it before each use. Here this would look like this:

```
mov t3, [rbp+s2]
mov t1, t3
```

Here the temporary is not needed at all—we can just load to the right register directly:

```
mov t1, [rbp+s2]
```

Similarly if `t1` was spilled and a store was needed, we can omit the copy:

```
mov [rbp+s1], t2
```

However, if both are spilled we need both a load and store, though the copy instruction can be deleted and a common temporary has to be used:

```
mov t3, [rbp+s2]
mov [rbp+s1], t3
```

And if we are able to assign `t1` and `t2` to a common memory location, then no instruction is necessary at all.

3.8.2.3 Interaction with instruction selection

As we have seen, the process of spilling needs to insert new instructions which together form the so called ”spill code”. In simplest scenario they just load and store the value, but better code can be achieved if the spill code is inserted with more thought than just load from memory and store to memory. But both of these problems—selecting the instructions to use for operations and choosing best ones in the current context is exactly the job of instruction selection.

In principle, register allocator shouldn’t care about the instructions. It should only care about their effects on the registers. The result of register allocation should be the assignment of register to virtual registers. Since spills can be necessary, which requires insertion of new instructions, we have to decide how this spill code will be handled. The basic options are the following, and mostly depend on the chosen register allocation technique:

1. Insert spill code in the register allocator pass.
2. Return the list of spilled virtual registers, and expect to be called again with code transformed to include the spill code.

The first option can make the register allocator depend on the target architecture—it now needs to know about the current target, its instructions, their meanings and how to insert them. On one hand, register allocation is already in the ”back end”, where we expect to handle things on the level of the target, and generally hope to take advantage of that by, for example, doing optimizations specific to the particular target. On the other hand, as mentioned previously, machine independent representations of (machine dependent) instructions are possible, thus spill code insertion *could* also be made machine independent similarly just like the register allocation.

The second approach also makes the register allocation process pure in a sense. The register allocator never modifies the input, its result is a either a mapping of virtual registers to physical registers, or a list of virtual registers to be spilled. Though it still has to be decided on how to insert the instructions. Some instruction selection mechanisms which ultimately depend on the middle end IR, are not suitable for inserting and optimizing spill code, since we are already in the low level back end IR. Tree or DAG based instruction selection mechanisms may also not be directly applicable—we may no longer be using trees or DAGs for representing the machine code, especially after instruction scheduling, which sets the order of instructions in stone. On the other hand peephole optimization is a great fit for improving inserted spill code. The inserted spill code can be very naive, and peephole optimization run on the spill code and its surroundings can make improvements. This is especially likely if we are able to find patterns which spilled code creates, such as in the example in the previous subsection (3.8.2.1).

The obvious downside of the "pure register allocation" approach is, that it has to start over with register allocation if any spills need to be made. The first approach seems more suited to register allocation where we want a self contained fast single pass register allocation.

The approaches used for spill code insertion are usually very connected to the core principle of the register allocation algorithm at hand, and choices of some approaches are discussed in section 3.8.4.

3.8.3 Concepts

The terms used in the previous sections about register allocation were not properly defined, and used a bit vaguely. The intention was to practically show the problems register allocation tries to solve and what it needs to do to solve it, which includes optimizations that try to make the register allocation process more optimal. In this section, we try to more properly introduce the terms used for concepts connected to register allocation.

One thing that has to be noted is the name "register allocation" itself. We mentioned that register allocation is meant to map virtual registers to physical registers of the target architecture, but the process isn't always so direct, and sometimes it makes sense and brings benefits to split this process into two parts, which really define what we mean by these names:

1. *Register allocation.* In a narrower sense, by register allocation we mean the process of making sure that each virtual register can be assigned a physical one. At this point, we may not care too much about which one, but we care about spill code, because that is what allows us to fit into the limited amount of registers available.
2. *Register assignment.* Assignments follows allocation—now that we can map every virtual register to at least one register, we choose the concrete one. Although this seems much more simpler than the allocation part, in practice register assignment is also very important, because we have seen situations where some assignments lead to better code, for example where the source and destination of a move instruction are assigned the same register, the move instruction can be eliminated.

Some register allocation algorithms intertwine both parts and don't split them. Some algorithms strictly separate these concerns. In general simpler algorithms usually merge both of these parts, while more complex algorithms try to take advantage of attacking each of those issues separately to reach more optimal results. But this distinction is of course not definitive.

3.8.3.1 Liveness

Already in previous sections we hinted that we may reuse a physical register if the virtual register occupying has no further use. Liveness is the property that captures this formally.

Definition 3.1 Liveness. A virtual register is *live* at a program point if it *may* be used in future.

The definition of liveness captures the "not used further" aspect that we found beneficial for register reuse. If at a program point a virtual register stops being live (becomes *dead*) the physical register allocated to it becomes available. The definition of liveness carefully says “may be used in future”, because in general it is undecidable whether a virtual register *will* be used.

3.8.3.2 Interference

Interference captures the fact that registers are live at the same time, and thus cannot be allocated the same register (nor coalesced).
Definition 3.2 Interference. Two virtual registers *interfere*, when they are simultaneously live at some point in the object program. [Chaitin et al., 1981]

3.8.3.3 Live ranges

It is usually not desirable to allocate registers for *variables*. We prefer to assign registers separately for each distinct definitions and its uses.

This is in principle very similar to SSA form, where we also want to split multiple assignments to each variable into multiple entities. In register allocation, the main advantage is, that the “splits” may be assigned physical registers or spilled separately.

The splits are known by different names in literature: *names* [Chaitin et al., 1981], *webs* [Muchnick, 1997], and *live ranges* [Cooper et al., 2004]. Live range is the often used name, because the splitting exactly corresponds to a technique which is able to introduce additional splits: *live range splitting* (see section 3.8.3.5).

Since we are mostly concerned with translation out of SSA, our virtual registers practically correspond to live ranges. In later text a mention of a virtual register should also be taken to mean a *live range*.

3.8.3.4 Coalescing

Coalescing in register allocation tries to assign virtual registers a common physical register. Usually we only care about coalescing where it is beneficial. In the low level these are the situations which may manifest as copies from one register to the other—if the two registers are allocated the same physical register, the copy is redundant and can be optimized out.

In practice this is connected to several high level concepts that generate low level copies:

- Translation from three address code to two address code. In two address code the first operand is the same as the destination. When translating three address code to two address code, we need to introduce a copy to not clobber the first operand in case it *lives out*. For example this three address code instruction:

  ```
  sub t1, t2, t3
  ```

  Can be translated to:

  ```
  mov t3, t1
  sub t3, t2
  ```

- SSA deconstruction (section 3.5). Effects of ϕ instructions are usually modelled with copies in predecessor blocks. For example the following instruction in block 3, which merges virtual registers `t1` and `t2` into virtual register `t3`:

  ```
  phi t3, t1, t2
  ```

  May translate to the following instructions in the two predecessors blocks:

```
// predecessor 1
mov t3, t1

// predecessor 2
mov t3, t2
```

Note that just these copies may be insufficient in some situations, see section 3.5.

- Live range splits. A virtual register can be split into multiple virtual registers, and copies are introduced to connect them. These can include splits due to register constraints (including calling conventions). See section 3.8.3.5 for more details.

  As coalescing is the exact opposite of live range splitting, virtual registers that are the result of live range splitting should not be carelessly coalesced, since that would practically undo the splits.

The removal of the move instructions itself is a simple task that can be left to the peephole optimizer. The real role of coalescing in register allocation is to try to allocate move related nodes to the same register. But too much coalescing can be counterproductive—choosing to allocate two virtual registers to the same physical one means that effectively the registers are combined into one. This means that their interferences add together and it becomes much harder to find a common register.

3.8.3.5 Live range splitting

Live range splitting is essentially the opposite of coalescing. By splitting one virtual register into multiple, we hope to achieve better register allocation, since the split virtual registers can be allocated separately and they can be much less constrained (have less interferences) than the original virtual register.

A big disadvantage of our model of register allocation being a mapping from virtual register to physical registers is that a virtual register can be assigned only one physical register. Live range splitting essentially remedies it, because the splits are allocated separately.

Another advantage is that, in the simple spilling implementation, every use and definition of a register is replaced accordingly by a load or store.

Naive spilling of an entire virtual register is simple to implement, but doesn't consider the nature of the register's use. For example the virtual register may see significant use in the beginning of the procedure and then in the end of the procedure and not be used much otherwise. Such virtual register will interfere with practically all other virtual registers, since it is live during the whole function. Splitting it to three registers (beginning, middle and end of the procedure) not only allows different physical registers to be allocated to it in the different parts, but it also allows *independent* spill decisions. For example, it can be spilled during the middle of the function cheaply with only one load and one store. The splits here would mean splitting say `t10`:

```
...
mov t10, ...
...,  t10

[...] // a loop

..., t10
```

To `t11`, `t12` and `t13`:

```
...
mov t11, ...
```

```
...,  t11

mov t12, t11
[...] // a loop
mov t13, t12

..., t12
```

Splits can be especially beneficial around loops, since spills of any values used inside loops are costly. Even a lot of shuffling of registers and memory can be beneficial if a loop is executed many times. Moving expensive operations out of the loops has been the task of code motion in the middle-end and instruction scheduling in the back end. Since register allocation generally follows them, we should do our best to not hinder it.

If the splits turn out to be unnecessary, they can be coalesced away. Though introducing too many splits can reduce the chance of coalescing in general—coalescing has to be careful about splits in order to not undo them carelessly, additionally many register allocation techniques avoid excessive coalescing.

Spilling can be seen as a very primitive form of live range splitting, which introduces a new virtual register for each use and definition and spills all of them. Splitting the uses and definitions manually, and spilling them only when needed may prove to be better, since even in case the copies across the splits don't get coalesced, they are presumably cheaper than touching the memory.

Some allocators are able to rethink the register allocation of a virtual register at its every use or definition. This means that the algorithm can essentially perform live range splitting everywhere. Other algorithms are not able to do splitting "online" and require it to be done as a preprocessing step, in that case live range splitting can be even more important.

3.8.3.6 Register constraints

Some instructions require the arguments to be in particular registers or produce values in particular registers. Typically, there are two such constraints:

1. One concrete register has to be used.
2. One register belonging to a particular class can be used.

Register classes are discussed in section 3.8.3.7. Here we discuss only constraints which require one particular register.

The x86-64 architecture has two notable examples where concrete registers have to be used for operands or results:

1. *Shifts.* Shift instructions where the shift amount is not an immediate value, require the shift amount to be in the `cl` register (that is, the low 8 bits of the `rcx` register).
2. *Long multiplication and division.* On x86-64 the only available division instructions (for signed and unsigned division) divide a 128-bit number by a 64-bit value.[1] Upper 64-bits of the 128-bit dividend are expected in `rdx`, the lower 64-bits in `rax`. The division instructions store the remainder in `rdx` and quotient in `rax`. The divisor can be a (64-bit) register or memory location.

But there are restrictions imposed not necessarily to the instruction set itself, but by *calling conventions* of the platform. For example on x86-64 Linux the System V

[1] As with other instructions on x86-64, the 32-bit variants are available. So for example, the long division divides 64-bit number by a 32-bit number, but it still operates on two registers (`eax` and `edx`).

ABI prescribes that the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9` are used for passing parameters and `rax` and `rdx` are used for return values.

Register allocator needs to conform to these requirements. But if the physical registers are allocated for the entire lifetime of the constrained virtual registers, then there can be conflicts if there are multiple such constrained contexts and the virtual registers interfere. For example, if there are the x86-64 shift instructions:

```
sal t1, t2 // t2 needs to be allocated to cl
sar t3, t4 // t4 needs to be allocated to cl
```

Here `t2` and `t4` both need to be allocated to `cl`, but they are live at the same time (interfere), so they cannot be assigned the same register. This is the case regardless of the number of available registers, so this is a problem of the *register assignment* part of register allocation. Spilling can save the situation, for example spill of `t4` helps if `t2` isn't needed after this snippet of code:

```
sal t1, t2 // t2 = cl
mov t5, [rbp+s4]
sar t3, t5 // t5 = cl
```

Spilling both of course also helps, but spill of `t2` alone doesn't, because the newly introduced temporary (`t5`, constrained to `cl`) would still be alive at the same as `t4` and thus they would interfere:

```
mov t5, [rbp+s2]
sal t1, t5 // t5 = cl
sar t3, t4 // t4 = cl
```

The problem with spilling here is, that it is mainly used as means of reducing register pressure. Because of that spilling heuristics are meant to spill virtual registers that are e.g. not going to be used for the "longest" (which frees a physical register for the longest time) or which interfere with a lot of other virtual registers (so the other virtual registers have much higher chance of being allocated themselves). Nothing usually makes constrained registers good spill candidates, and it shouldn't, since because of the constraints the virtual registers *need* to be assigned.

Live range splitting is a much better choice for handling constrained registers. By introducing a new virtual register (and a copy to it) for the short time of the constrained use, the constrained use doesn't have any chance of interfering with other constrained uses and is so short lived that it even isn't a plausible spill target (see section 3.8.2). In our example, it looks like this:

```
mov t5, t2
sal t1, t5 // t5 = cl
mov t6, t4
sar t3, t6 // t6 = cl
```

Here indeed `t5` and `t6` don't interfere and are unspillable. If, like we assumed earlier, `t2` doesn't *live-out*, then the best assignment would assign `t5` the `cl` register (low 8 bits of `rcx`), like this:

```
mov rcx, rcx
sal rax, cl
mov rcx, rdx
sar rbx, cl
```

Here it was possible to keep all values in registers. The first copy can be easily optimized by subsequent peephole optimization pass.

Live range splitting is good solution for making register constraints not constrain the register assignment much. The same ideas that apply to live range splitting apply also here, in particular allocators which are unable to perform live range splitting on demand should split constrained uses beforehand and coalescing may be able to remove the copies if it turns out they are not needed.

Calling conventions also dictate the coordination of registers between the *caller* (calling function) and the *callee*. Both want to use machine registers (and ideally all of them), but if callee uses the registers, it overwrites values the caller has stored. For that purpose registers are classified as either caller-saved (caller has to save the registers, if it uses them) or callee-saved (callee has to save the registers, if it wants to use them).

The fact that a register is caller saved means that when a function performs a call, it has to pessimistically assume that the callee changes all the caller saved registers.[1] This can be modelled like an register constraint on the call instruction. For example on x86-64, where for example `rax`, `rcx` and `r11` are caller saved we can make the `call` instruction *define* the registers:

```
call f % defines rax, rcx, r11 and others
```

Coincidentally `rax` is at the same time used for passing the return value, so it would have been defined by the call instruction already. But `rcx` for example is used for parameter passing and is thus as mentioned above *used* by the call instruction to model that. Adding definition of `rcx` to the call means, that the caller shouldn't expect the `rcx` register to be preserved by the caller. While this model works well for *correct allocation*, a lot of registers are caller saved and virtual registers that *live through* the call can not be allocated to them. Callee saved registers are needed in that situation. But for example on x86-64 there are 9 caller saved registers and only 7 are callee saved, and out of those 2 are usually reserved for special purposes (stack and base pointer). Having only a few available registers for values living across calls means very high register pressure, which will have to be mitigated by spills. Spills are correct and needed here, since if the registers are reserved for the caller, we don't have any other choice for storing values than memory. But naively spilling virtual registers everywhere just because of high register pressure at a call site is not ideal. Once again this is a place where live range splitting helps—just like splits around loops were useful, splits around calls can be useful for minimizing damage implied by spilling virtual registers at every use or definition.

Callee saved registers can be modelled through uses and definitions as well. Having definitions of callee saved registers at the entry point of a function and uses at the exit point (return instruction) models the fact, that the callee saved register has to be preserved for the entire duration of call. This not only requires physical registers to be somehow represented, but also blocks the callee saved register for the entire duration

[1] The callee doesn't necessarily change any of the caller saved registers, but it is allowed to do so. Calling conventions are general, and don't try to specialize for some cases. They are the interface functions need to conform to, and allow code produced by different compilers to cooperate together. But if a compiler is sure that the function doesn't have to conform to the interface (perphaps because the function is `static` and thus not callable by from other separately compiled modules), then it can can try to do better by employing whole module register allocation, which considers register allocation even across the calls. Often though a much simpler technique helps with calling convention constraints—not performing any calls at all! Inlining the called function into the call site means that the registers used be the called function are allocated as part of the function procedure and the calling convention constraints don't apply at all. Leaf functions (functions not calling other functions) are especially good candidates for inlining especially because they don't impose calling convention constraints themselves.

of the function. Live range splitting is again very useful here, because we can introduce virtual registers for holding the values in callee saved registers during the function, e.g. if we consider just `rbx` and `r12`:

```
mov t20, rbx
mov t21, r12
[...]
mov rbx, t20
mov r12, t21
```

This is much better, since the virtual registers can be spilled, which frees up a callee saved physical registers, which can possibly be used by many (short lived) virtual registers or perhaps just one virtual register with more uses and definitions which would be more expensive to spill. The virtual registers introduced for callee saved registers are in fact ideal spill targets—there is only one definition and one use, both outside of any loop. Spilling them could look like this:

```
mov [rbp+s20], rbx
mov [rbp+s21], r12
[...]
mov rbx, [rbp+s20]
mov r12, [rbp+s21]
```

Here we refer to some abstract “stack slots”. If the stack slots are chosen well, `push` and `pop` instructions can even be used for realizing those spills:

```
push rbx
push r12
[...]
pop r12
pop rbx
```

Which essentially gets us the code one would use to free up callee saved registers. But modelling it through register constraints can nicely take care of only using the callee saved registers when beneficial and so it is more flexible.

3.8.3.7 Register classes

On real world architectures almost always not all registers are equal. For example on the x86-64 we have general purpose registers as well as vector registers (and also the floating point registers, which are not real registers per se and because they are largely obsolete we will not consider them further). Additionally, many actual *hardware registers* (like `rax`) have parts which are accessible with different *names* (`eax`, `al`, `ah`).

We call the set of registers which are interchangeable in some context *a register class* and the fact that a multiple names may access the same register *register aliasing*.

Most architectures like the x86-64 have apart from general purpose register class an entirely independent floating pointer (or vector) register class. Dealing with independent register classes is relatively straightforward—register allocation can be done mostly separately. Though general purpose registers have a special status—they are used for address calculations and *spills* address memory. In practice however, spill locations are usually on the stack and CPUs are able to access relatively to the stack pointer (or base pointer), so spills don’t influence general purpose register allocation.

A problem occurs when occurs when register classes *overlap* and are not independent. Aliasing is one thing that makes register classes overlap—for example, on the x86-64 64-bit register (`rax`, etc.) overlap with 8-bit registers (`al`, etc.). This brings many problems, since allocation of `rax` for example "blocks" both `al` and `ah`, while allocation of `ah` blocks `rax`, but not `al`. Register class hierarchies, where the classes all not all independent, require special handling.

The paper by Smith [Smith et al., 2004] is perhaps still the definitive treatment of register classes in register allocation literature. We used similar terminology as them in this section and further consider their approach in the next section.

3.8.4 Techniques

We have already seen a few things that can distinguish different register allocation algorithms:

- Handling of spilling (section 3.8.2.1),
- Split or no split of allocation and assignment (section 3.8.3).

But there are others:

- Scope: *local* vs *global* vs *interprocedural* vs *whole program* algorithms. Local algorithms operate on singular basic blocks and use only information local to the basic block to decide on register allocation. The limited scope makes the algorithms generally simpler and produces worse results then global register allocation, which allocates registers to whole functions. Global register allocation is global in the sense that *all* basic blocks are considered at the same time. The analysis is more complex, since it has to handle control flow. Even techniques for allocating registers across function calls and whole programs exist. These can be less practical in practice, where functions may be required to conform to a *calling convention*, which specifies how arguments should be passed in function calls, what registers are preserved by calls and where will the return values reside. We will not discuss techniques operating in larger scopes than *global* (whole function, all basic blocks).
- *Quality* vs *speed*. With no restrictions on time, we ideally would like to achieve *optimal* register allocation. With the right definition of optimal, it can be possible, but due to the difficulty of the register allocation, this approach is bound to be too slow (in *compile-time*), although it would produce code that would be fast (in *run-time*). In code compiled ahead of time, we can probably justify spending more time on compilation to achieve better run-time, since it is expected, that the program will run for some time and that the investment will return.

  On the other end of the spectre we may want a register allocation algorithm that runs very fast (due to constraints on compile-time), but in that case we can't expect good results (i.e. code that has fast run-time). This can be interesting for *Just-in-time* (JIT) compilers, where the compile time is part of run-time and hence it is not possible to spend much time on optimizations, because it is possible that they wouldn't pay off (though they could, we don't know ahead of time).
- *Control flow sensitivity.* Some global algorithms may completely disregard the actual control flow of the program and just use (global) liveness and/or interferences to do register allocation and assignment. But use of control flow information in an algorithm is likely to steer it to better results—spills in hot or nested loops are undesirable. Control flow sensitive register allocation (not assignment) may for example even try to purposefully do spills or splits before loops to make more registers available in loops.

3.8.4.1 Local top-down and bottom-up

Two of the most basic algorithms for register allocation are described by Cooper and Torczon [Cooper et al., 2004]. They operate only on single basic blocks, but form a good baseline to improve upon. Also, surprisingly, their ideas of how to handle spilling have their equivalents in more powerful algorithms.

Both algorithms investigate uses and definitions of virtual registers inside a basic block, allocate some map some virtual registers to physical registers and spill the others. They differ in how they choose which virtual registers to spill:

- *Top-down.* In the top-down view those virtual registers which are used most often (i.e. their number of uses and definitions is the highest) should be the ones that get assigned registers, others should be spilled.

  While this is simple to implement, there are glaring problems. The algorithm is not able to reuse registers—since it maps virtual registers to physical registers one-to-one, it is not able to reuse a physical register once a virtual register becomes dead.

  Also, most instructions require at least some arguments to reside in registers, but it can happen that top-down register allocator spills all used and defined virtual registers of a particular instruction. To solve this, sufficient number of registers has to be set aside and not be allocated, they will be used for realizing loads of uses and stores of definitions of spilled virtual registers. This of course makes the allocation results even worse, because only a lesser number of registers are available for allocation, and spills may be introduced just because some registers are reserved for spill code realization.
- *Bottom-up.* In the bottom-up approach instructions are investigated in order, one by one, and registers are allocated to supply the demand of each particular instruction. In general, each instruction is an operation with multiple input registers and multiple output registers. Hence for each instruction the algorithm ensures that input virtual registers are allocated into physical registers and allocates registers for output virtual registers.

  It may seem, that since the algorithm operates on a single basic block, that each use of an virtual registers should have a preceding definition, which should be the one, which allocates register for it, and that allocation of registers for arguments is not necessary. But it is necessary for handling spills—allocation of a physical register (whether for input or output virtual register) may find that none of the registers is free, so it has to choose one of the assigned registers, and spill it, by moving the value from that register to memory. Later the spilled register may be needed again, and so it has to be assigned register again and the previous value has to be reloaded from memory into the new register. The newly allocated register doesn't have to be the old one. This is a great advantage of this approach over *top-down*, while spilling, it is able to effectively split a live range and allocate it different physical registers or memory locations.

  Because the algorithm considers each instruction, it is able to much better deal with machine constraints. For example, if an instruction needs its operand to reside in a particular register, or puts the result in a particular register, then the allocator may just forcibly allocate that particular register. One example of such constraint are that of the shift instructions on x86-64, which require the shift size to be specified in the `cl` register:

  ```
  shl t1, t2 % t2 has to be allocated to cl
  ```

  If we suppose that `t1` already resides in a register (say `rax`), and `t2` is already in `rcx` (the register of which `cl` is the lowest 8 bits), then the allocator doesn't have to do anything:

```
shl rax, cl % t2 has to be allocated to cl
```

If t2 is assigned say the rdx register, and the value in the rcx register is no longer needed, just a copy is sufficient:

```
mov rcx, rdx
shl rax, cl
```

If however, the virtual register which occupies rcx (say t3) is live after the instruction, then we need to find it a new register. And if conveniently t2 (the shift amount) is not needed after the shift, then we can just reuse the newly freed rdx register by swapping the registers:

```
xchg rcx, rdx
shl rax, cl
```

And so on. Similarly we could deal with platform calling conventions. For example, if the called function needs arguments in registers rdi and rsi, then we might just forcibly allocate them. If the function call doesn't preserve other registers (like rax or rcx), these registers should also be forcibly allocated—though the call-site will not use them for anything, the called function might, and hence we need to preserve values in them, and using the "allocate a register" mechanism, we elegantly also handle the necessary spills.

When a physical register is needed and none is available, one has to be spilled. Good choice is to spill the virtual register whose next use is the furthest away [Cooper et al., 2004]. This is akin to Bélády's MIN algorithm for page replacement [Belady, 1966]. The benefit of the approach is, that if we need to spill, the register we free up will be available for other purposes for the longest time, hence it will hopefully prevent other spills.

The bottom-up algorithm has to do two passes over the code—first one to derive liveness information, second one to actually do the allocation. Liveness information provides the information necessary for choosing spills.

An interesting twist to the bottom-up algorithm described by Mike Pall [Pall, 2009]. He does the allocation in a single pass over the code in SSA form, though in *reverse*. In programs in SSA form SSA values naturally correspond to live ranges, and reverse order is natural for computing liveness. Pall's algorithm essentially combines register allocation with the liveness computation. While iterating in reverse order definitions are processed first, while uses are processed next, contrary to the bottom-up algorithm. Also contrary to the bottom-up algorithm, where *definitions* was what derived the assignment, it is the *uses* that drive the assignment in this algorithm. When a first use of a virtual register encountered, it is allocated a register, and when (the only) definition of a virtual register is reached, then it is freed. This is often better, since more often the uses are constrained by machine constraints, so by discovering the uses first, the allocation can be targeted more easily.

The problem with all these three approaches is that they are too local. Their versions as presented above work only in a single basic block. Extensions to global (whole-procedure) allocation are possible by using memory—a simple extension does register allocation on each basic block separately and all virtual registers are stored at the end of each block and loaded back at start of a each block. This still only requires only local analysis, but produces very inefficient code. It is possible to improve this by performing global liveness analysis and to store only virtual registers that live-out and to load only live-in virtual registers. Though at the point where global analysis is feasible, some of the global register allocation algorithms is probably feasible as well.

3.8.4.2 Linear scan

Poletto and Sarkar [Poletto et al., 1999] introduced o called *linear scan* register allocation. It can be seen as an extension of the bottom-up approach described in the previous section (3.8.4.1). The canonical version of the bottom-up is able to allocate registers only for a single basic block, because it depends on many of the linear aspects of basic blocks, such as that the instructions are ordered, live ranges are also ordered and it can be determined which is "furthest away", so that something akin to Bélády's algorithm [Belady, 1966] can be used. Linear scan register allocation extends the bottom-up approach to perform global (whole procedure) register allocation by imposing an ordering over the instructions by (globally) numbering them. This ordering induces a linear sequence, where live ranges can be represented as simple intervals starting at the number of the first instruction where the virtual register is live and ending at the number of the last instruction where the virtual register is live. This essentially makes the procedure into a single "basic block" on which something akin to the bottom-up allocator can be run.

But the algorithm described by Poletto instead operates on the live intervals. The algorithm orders the intervals by increasing start point and iterates over them. The algorithm effectively iterates over the starts of live ranges, keeping the set of *active* intervals (those whose start is *before* the start of the current interval, and end *after* the current interval). For each encountered interval, those intervals in the active set which end before the current interval's start are expired and their registers freed, and a new register is a allocated from the pool of free registers for the current live interval. If the number of intervals live at some point exceeds the number of available registers a live range needs to be spilled. Poletto and Sarkar choose to spill the live range which ends furthest away—if that live range is the one which is currently being allocated, it is not allocated a register, but a memory location instead, otherwise the current live range is assigned the register of the spilled interval and the spilled interval is assigned a memory location.

There are many problems with linear scan. It has been designed as a fast and simple alternative to graph coloring register allocators (see 3.8.4.3), mainly for JIT compilers which value greatly run-time of *the compiler* and can sacrifice run-time of the compiled code. The relatively poor allocation quality is intentional.

The reasons for the poor quality is that live ranges are really really imprecise, since this simple live ranges don't represent live ranges truthfully—there may be many instructions in the middle of the interval, where the virtual register is not live. In fact trivial live range $[1, n]$, where n is the number of instructions is correct for each virtual register, but of course produces unsatisfactory allocations. Another problem is, that unlike the bottom-up approach linear scan has more trouble with handling of spilled code as well as machine constraints. This is because virtual registers (live intervals) are assigned either a register for their entire duration, or a memory location. For use of the memory locations in instructions registers have to be used, and a few registers would have to be set aside for that (like with the top-down allocator from section 3.8.4.1). Basic form of linear scan doesn't handle machine constrains at all

While in some sense linear scan register allocation can be seen as a extension of the bottom-up register allocator, it suffers from many of the issues of the top-down allocator. Some of the follow ups on linear scan are much better suited to practice. For example in [Traub et al., 1998] they are able to additionally deal with holes in live intervals and can assign a multiple registers to a single live range (at different times) and other approaches are able to better handle machine constraints and use properties of SSA form [Mössenböck et al., 2002, Wimmer et al., 2010].

3.8.4.3 Graph coloring

Even though the idea of using graph coloring for register allocation is older, first notable use of the technique is by Chaitin [Chaitin et al., 1981, Chaitin, 1982]. The core of the idea is to construct an interference graph from the interferences of virtual registers—all virtual registers become nodes in a graph and there is an edge between virtual registers if and only if they interfere. Then on this graph we aim to find a *coloring*—mapping of nodes to colors, such that no neighbouring nodes get the same color. In our case the colors ultimately constitute the machine registers. Because edges designate interference, it is guaranteed that no virtual registers that interfere are assigned the same physical register. If we have k machine registers available, then we are looking for a k coloring—the coloring needs to use at most k colors (registers).

By reducing the register allocation to graph coloring we may seemingly not gain much, since graph coloring is an NP-complete problem[1]. However in [Chaitin et al., 1981] they rediscover a technique that can simplify and make graph coloring practical for register allocation. It is based on the observation that a node which has fewer than k neighbours can be always assigned a color distinct from all of its neighbours. Because the node has less neighbours than there are available colors, even if all neighbours used different colors, there would still be a free color left. This simple, yet important observation is the base for their and derived techniques.

Since incorporating a node with degree (number of adjacent nodes) less than k (a so called *insignificant* node) into an already colored graph is easy, initially we do the opposite—remove from the graph all insignificant nodes, such that later, in the reverse process we can add them back to the graph and color them trivially. Removing low degree nodes from the graph causes the degrees of neighbouring nodes to decrease as well and may thus lead to more simplifications. In Chaitin's algorithm this phase is called *simplify* and the removed low degree nodes are pushed onto a stack. In the final stage, called *assign*, the nodes have colors assigned in the reverse order simply by popping them from the stack and assigning them a color not used by any of the already colored neighbours in the now being rebuilt graph. Use of this heuristic is not all saving—it is possible that after after simplification (removal of low degree nodes) there will still be high degree nodes left. In that moment, push of any of the remaining nodes on to the stack, could mean that there won't be a color left for it. Chaitin's solution is to calculate spill costs of all the remaining high degree nodes, choose the one with the lowest cost, mark it as to be spilled and remove it from the graph. Due to the removal, the simplification process may find more simplifications, otherwise another spill decisions may be made. The removal of the to be spilled node from the graph simulates its replacement by loads and stores, which although will introduce new virtual registers, they will have very short live ranges, with (hopefully) much less interferences, so it suffices as an approximation.

Spill of any node means that the code needs to be updated with spill code, and the register allocation process repeated. Since the program is now different, and new pseudoregisters were introduced to accommodate spill code, liveness analysis and building of interference graph have to be repeated as well. Then simplification can be tried again. This entire process is tried until the simplification is able to reduce the graph to an

[1] In [Chaitin et al., 1981] authors argue further that register allocation is also NP-complete, this has since been disputed [Bouchez et al., 2007].

empty graph, which is trivially colorable. Since each iteration is very expensive, it is important that there can be multiple spill decisions made in a single *simplify* run, this way the process can often finish in 1 or 2 iterations, if the first iteration successfully finds all nodes that need to be spilled and the second iteration finalizes the assignment. Being able to spill only one node on each iteration would mean that graphs with many high degree nodes would need *many* expensive iterations to finish. Proceeding from *simplify* only after all spill have been handled makes it possible to push only low degree nodes, guaranteeing that in the reverse *assign* stage, every node popped from top of the stack will have at least one free color.

But, it is possible to do better. The fact, that a node has a *significant* number of neighbours doesn't mean, that it will be uncolorable in the *assignment* stage. There is a chance that the already colored neighbours will be assigned less than k distinct colors, in that case the popped node could still be colored even though at the time it was pushed it was significant. Because of this, in *simplify* we may optimistically try to remove and push high degree nodes on to the stack, instead of pessimistically spilling them. If in the assignment phase it turns out that there isn't a color left for the popped node, we spill it only then. We call these pushed high degree nodes *potential spills*, since they become *actual spills* spilled in the *assign* stage. This strategy is called *optimistic coloring* and was devised by Briggs [Briggs, 1992, Briggs et al., 1994]. Even in this strategy, it can happen that more than one (potential) spill will be necessary. Like with Chaitin's "pessimistic" spilling, the best possible node for potential spill is the one with the lowest spill cost—first potential spill will be processed last in the *assign* stage, and will encounter a more complete interference graph, than potential spills pushed later, which are assigned colors earlier. Like before, we want to capture all *actual* spills, before we repeat the whole process with spill code inserted. To do this, in the *assignment* stage we don't allocate the actual spills any color, just mark them for spilling and proceed. This can make more some nodes neighbours of actual spills colorable, since by not coloring the actual spill, they effectively have one less interfering node. Like with Chaitin's spilling in *simplify* stage, this approximates the actual effect of spilling, which splits a single node into many temporaries whose interferences are more local and hopefully easier to deal with.

As we have seen before (in section 3.8.2), spilling can always be necessary, reducing the register allocation problem to graph coloring doesn't change that. No matter how k is big, there are always graphs which need more registers to be colored successfully. Even an exact graph coloring algorithm that tries all possibilities can fail to find coloring because of this. Spilling is thus *not* only due to Chaitin's heuristic. Though the heuristic even with Briggs' optimistic coloring can introduce more spills than a more exact algorithm would.

The interference graph is a really great data structure, because apart from being able to represent the "live at the same time, and thus unallocatable to the same register" constraints, it can express also other restrictions. Machine constraints and calling conventions can both be modelled by interferences (edges in the interference graph) if we also add physical registers as nodes. For example, we can force a virtual register node to be allocated to a particular physical register by making it interfere with nodes corresponding to all the other physical registers. This is often called *precoloring*. In fact, since we need to be careful about not accidentally allocating physical register a different physical register, we need to make all physical registers interfere with each other, this way they are all guaranteed to be allocated their color (register). But these additional constrains can lead to uncolorable ("overconstrained") graphs if the live ranges of precolored registers are too long. Since graph coloring maps each virtual register to a single physical register, it needs the precolored live ranges to be short and non-interfering, which can be done with *live range splitting* (see section 3.8.3.5).

But avoiding uncolorable graphs with splits means a lot of copy instructions, which if allocated different registers, will not be optimized by peephole optimization and thus can incur significant unnecessary overhead. This increases the need for *coalescing* (see section 3.8.3.4). Chaitin already realized the need for coalescing. His solution [Chaitin et al., 1981] was to coalesce every *copy-related*, *non-interfering* pair of virtual registers in a pass called *coalesce*, before simplification. A pair of virtual registers `t1` and `t2` is copy related when there is a copy (move) instruction between them (i.e. `mov t1, t2` or vice versa), which captures the goal of eliminating these moves. The virtual registers have to be non-interfering, since otherwise the coalesced node would be uncolorable, and also, since the temporaries interfere, they wouldn't be assigned distinct colors, and elimination of the copy wouldn't be possible anyways. Because of the non-intefering criterion, we have to be careful not to create artificial interferences for the operands of a copy instruction, for example `mov t2, t1` alone shouldn't imply that `t1` and `t2` interfere! Chaitin essentially does coalescing everywhere where it is possible and where it *might* be beneficial. Because of its nature, this form of coalescing has later become called *aggressive*. The problem with it, is that the node created by coalescing two virtual registers has interferences of both of the former nodes, notably this means that the node's degree will be the sum of the two degrees and such nodes easily become significant ("high degree", not trivially colorable). High degree nodes are problematic in the following *simplify* phase, because apart from being blocked from simplification themselves, they prevent simplifications on a high number of other nodes. Often this means spills of these high degree nodes. Aggressive coloring can make colorable graphs uncolorable, and depends on spilling to make the graph colorable again. Since a spill of a node essentially splits the node into many low degree nodes, this effectively undoes coalescing, but also adds memory operations that weren't there originally.

Briggs improved on this by employing so called *conservative coalescing*. Instead of coalescing all nodes that can be coalesced, he uses a filtering heuristic, which allows only those coalesces, that can't make the graph uncolorable. The filtering is done using a heuristic, because exactly predicting the effect on colorability is a hard problem and would be too time consuming. Since the effect of aggressive coalescing can be so severe, the heuristic was made conservative, i.e. it never allows coalescings which would make the graph uncolorable, but may also not allow coalesces that would be perfectly fine. The heuristic says that `t1` and `t2` can be coalesced only when the merged node `t12` would have no more significant (high-degree) neighbours, than k (the number of available registers). This implies, that after simplification of (low-degree) neighbours, the node will have at most k neighbours left, which makes it simplifiable itself. Though it is easy to imagine a situation where a lot of the neighbours have common neighbours, so simplification may do much better than conservatively assumed, and thus a lot of the moves remain uncoalesced.

Appel and George [George et al., 1996] found that for their use aggressive coalescing produced too many spills, while conservative coalescing was too conservative, i.e. there were are too many uneliminated move instructions left, even though coalescing would be fine. They suggest an improvement called *iterated register coalescing*. The idea is to still use only Briggs' conservative coalescing (to prevent making the graph uncolorable), but instead of doing all the coalescing upfront, they iterate the simplify and coalesce phases repeatedly. What is important is, that simplify precedes coalescing—this alone improves the coalescing phase a lot, since the conservative heuristic is based on degrees of nodes, and simplification can decrease them significantly (and Briggs' coalescing heuristic is too local to notice that otherwise). But importantly after coalescing there may be nodes which become insignificant. For example, if `t3` interferes with both `t1` and `t2`, and the two are coalesced into `t12`, then in effect `t3` loses a neighbour and it's degree is decreased, and it might just become insignificant ("low degree") and simplifiable (leading to more simplifications, which in turn might lead to more coalescings, etc.). While this may seem like a perfect positive feedback loop, it is important to recall, that coalescing two nodes creates a node of higher (even significant) degree.

Park and Moon note that even iterated coalescing can be too conservative and not combine nodes that could be safely combined. They also note that that the positive effect of coalescing explained in the previous paragraph is not to be underestimated. Their approach is called *optimistic coalescing* [Park et al., 2004], not to be confused with *optimistic coloring* due to Briggs [Briggs, 1992]. Park and Moon's idea is to do aggressive coalescing like Chaitin did, to exploit the positive effect of coalescing, but their improvement lies in being able to revert coalescing of a particular node, if it would have to be spilled. Briggs' optimistic coloring delayed actual spilling until the *assign* phase, since by then it may turn out that the concrete assignment isn't as unfavorable as it could have be just by judging from the interference graph and the simplification heuristic. Similarly Park and Moon moves decisions to *not coalesce* into the *assign* stage, and they are able to do better just because the concrete assignment is known. For example, in case a color is not available for `t12` (the result of coalescing `t1` and `t2`), it may be possible to find a color for `t1` or `t2` (or both, though it will not be the same color), which effectively undoes the coalescing. Though in practice, while Briggs' optimistic coloring improvement was simple addition and a sure improvement, optimistic coalescing and especially an efficient implementation is not simple.

Smith [Smith et al., 2004] provides an extension to graph coloring allocators based on Chaitin's heuristic. They generalize it to support complex *register class hierarchies.* They approach is general enough to handle all sensible (and thus actually existing) register class hierarchies, include x86-64's. Chaitin's heuristic compares the *degree* of node to the number of available registers (k)—if the degree is smaller the node can be simplified. With register classes the number of available registers is different per class (which is determined based on the uses of the virtual register) and degree no longer reflects the number of registers occupied by the neighbouring registers. The core of their idea is pretty simple instead of comparing the degree to the number of available registers, compare the maximum number of *registers denied by the neighbours* to the number of *registers available in the register class.* Like Chaitin's heuristic this is a simple comparison just analogously extended to register classes and reflects the fact that we can simplify (remove from graph) a register only if we can be absolutely sure that there will be a color left for it in the assignment stage. However, the maximum number of registers denied by the neighbours depends on the actual coloring—e.g. assignment of `rsi` denies only `sil`, while `rax` denies both `al` and `ah`. Investigating all colorings for each simplification is not feasible, however [Smith et al., 2004] describe a safe approximation, which is as easy to maintain as the current degree of the node, while being close to the actually possible maximum to not be overly pessimistic. The changes to the new version of "degree" are based on the classes of the neighbours and the decrements can be derived statically from the properties of the register class hierarchy.

Nice thing about Chaitin's scheme (and Briggs' improvement) is that even the introduced spill code with new virtual registers gets the same general treatment as other virtual registers - they are allocated by the next iteration of graph coloring, so although the spill code needs to be inserted separately, it is *not handled specially*. The price for this is that multiple expensive iterations may be needed to finalize the allocation. An alternative would be to (like with the top down allocator in section 3.8.4.1) reserve a few registers off the side and use them to perform the loads and stores around spilled variables. This could be used to rewrite the program into final form after just one iteration of the graph coloring register allocator. While this potentially saves multiple expensive iterations, it is less flexible than coloring the spill code in a new iteration. In particular, since the few spill handling registers have to be set off the side for the whole program, we are not able to assign them, so in fact we are looking for a k coloring for a smaller k than the number of available registers, which potentially means more spills by itself. On architectures like x86-64, where some instructions only work with certain registers there is another difficulty in choosing the on the side registers. If the registers needed by the constrained instructions are put off the side, they would prevent any allocation. But keeping them in the regular allocatable set would mean that they won't be available for handling the spills of the values constrained to such registers, the off the side registers would have to be used to somehow swap the values with the needed registers.

3.8.4.4 Graph coloring of chordal graphs

While graph coloring in general is an NP-complete problem, for certain classes of graphs, it can be easier. Notable example are chordal graphs, which have posses useful properties for efficient graph coloring. It turns out, that chordal graphs are the exact class of graphs for which exists a so called "perfect elimination order". Importantly for graph coloring, by assigning colors to nodes in the perfect elimination order, the graph can be colored with k colors in a single greedy pass—of course provided that the graph is indeed k-colorable. Similar greedy coloring pass was the *assign* phase of Chaitin's algorithm [Chaitin et al., 1981], there the node ordering was determined by simplifications based on a heuristic and colorability was ensured by spilling nodes. Perfect elimination order can be found in $O(n^2)$ time using the maximum cardinality search algorithm and *guarantees* optimal coloring. Chordal graphs also offer improvements for spilling, because like with perfect graphs of which they are a subset, the number of colors needed to color a chordal graph is given by the size of the largest clique. This is powerful, because it gives the possibility to do enough of spills or live range splits ahead of time, before actually starting with coloring.

The first application of these ideas to register allocation are due to Pereira and Palsberg [Pereira et al., 2005], who noticed that 95 % of interference graphs in the Java 1.5 library had chordal interference graphs (when compiled with JoeQ compiler). The algorithm proposed by them operates in a few independent phases:

1. *pre-spilling.* Spill code is inserted in order to decrease the size of the largest clique to k, which makes it k colorable.
2. *greedy coloring.* The graph is greedily colored without limiting the number of available colors.
3. *post-spilling.* If the number of used colors exceeds k, additional spills are done.
4. *coalescing.* Move related nodes are coalesced, if possible.

Both *pre-spilling* and *coalescing* are entirely optional—spills can be handled by post-spilling phase and coalescing is not a necessary port of any register allocation algorithm. But they both improve the quality of the generated code. In particular due to the properties of the chordal graphs described above, pre-spilling is a much better place for introducing spills, and if done properly, it is guaranteed that post-spilling doesn't need to do anything at all. Case when *post-spilling* comes into play are when the optional pre-spilling isn't run, or when *non-chordal* graph is being colored—Pereira and Palsberg noticed that the same algorithm can be used also for non-chordal graphs, though the register assignment is not optimal, it is competitive according to them [Pereira et al., 2005].

One of the important benefits of the algorithm is, that it isn't iterated, it finishes after running each phase only once. Though to be more precise, the post-spilling phase is not a single step—if more than k are used, all nodes of one color are spilled (transforming a graph using m colors to one using $m-1$), so this needs to be iterated until the only k colors are used. Though this is bounded, usually fast and may not be needed at all if *pre-spilling* phase is run.

While there are interesting similarities to Chaitin's classical approach, in particular being able to spill enough so that greedy algorithm can find a coloring, there are also interesting differences—in Chaitin's and derived algorithms coalescing is done *before* assignment, this can have both positive and negative impacts on colorability, and different variations approached it differently. Pereira's algorithm does all coalescing *after* assignment because after coalescing an interference graph can become non-chordal, though they report that in their experiments their approach does better in coalescing than Appel's iterated register coalescing.

While Pereira's algorithm can be used even for non-chordal interference graphs, following researched by e.g. Hack [Hack et al., 2006] showed, that programs in (strict) SSA form have chordal interference graphs. This is seems like an excellent result, because SSA form is great for middle-end optimizations and it simultaneously seems to be good for register allocation. Additionally SSA form allows much more efficient computation of the liveness property. The basis structure of Hack's algorithm is similar to Pereira's—do spilling before coloring to make the program k colorable, then color the graph in a perfect elimination order. However, there are substantial improvements: as they prove that interference is directly connected to the notion of dominance deeply associated with SSA, it is possible to derive the perfect elimination order from the control flow graph and dominator tree alone. Also, if we assume that copy propagation has been run before register allocation, the only coalescing that has to be done on SSA is the coalescing of ϕ node operands—by assigning the operands the same color as the phi node, SSA deconstruction doesn't have to insert any moves. However, in fact SSA deconstruction and coalescing have to be integral parts of their algorithm, since inserting arbitrary moves for SSA deconstruction could make the graph non-chordal (and in fact SSA incompatible) or increase register demand—which would be detrimental, because now SSA deconstruction is done *after* register allocation.

The fact that Hack's algorithm is able to take advantage of the many guarantees of SSA form, is also it's great disadvantage—it depends on the program on being in SSA form, which is not the norm. Usually, compilers do register allocation only after instruction selection during which concrete machine instructions are chosen to implement the behavior of the SSA-based intermediate representation. Keeping the ϕ instructions in the later stages of the compiler means that the algorithms have to respect *parallel copy* semantics of the ϕ instructions.

3.8.4.5 Reduction

Like with instruction selection, since register allocation is a very hard problem, it is possible to leverage existing efficient methods for solving other problems through a *reduction*.

The methods using reduction fully transform the register allocation problem to an instance of another problem, then use (often an external) generic solver and then translate the result back into a register allocation. This is a bit different from the graph coloring register allocators, which use ideas of graph coloring, but can still be seen as solving the register allocation problem directly—the coloring itself is only part of a scheme that consists of also spilling and coalescing, which are not handled as part of the graph coloring in an integrated fashion.

A notable example is reduction to integer linear programming by Appel and George [Appel et al., 2001]. The characteristics of their approach are separation of spilling, which is done upfront in the first stage, and extreme live range splitting, where they split at every program point and then do register assignment and coalescing in the second stage.

Scholz and Eckstein [Scholz et al., 2002] reduce register allocation to partitioned boolean quadratic problem and solve it using a heuristic running in near linear time.

Chapter 4
Design and implementation

This thesis looks into practical issues of taking a modern middle end intermediate representation and translating it to executable machine code of a real architecture, the x86-64. This is not something that hasn't been done before. The main benefits of this thesis are doing it as part of a TinyC compiler. TinyC is like C, but simpler in some aspects, though still keeping many challenges for implementations of middle ends and back ends alike.

In the NI-GEN course at FIT CTU, where TinyC originated, students write compilers from TinyC to *Tiny86*.

After careful consideration we set the following goals for the implementation of a TinyC back end for x86-64 and the runtime:

- The compiler should explore how TinyC features translate to constraints of a real architecture.
- Execution on real hardware (x86-64 CPU) and operating system should be possible.
- Advanced global optimizing techniques should be used.
- Simplicity of the compiler structure and of the code shall be one of the main goals, because the back end will serve as a demonstration of compiling TinyC to a real machine architecture.
- The source code should be readable by students of the NI-GEN course, who are learning about compiler back ends.
- The number of intermediate representations should be low. This will hopefully make the code more digestible and prevents much of the code being just translations from one IR to another.
- The back end should not have many external dependencies, since while they may simplify development and implementation, they hide details that are necessarily part of the implementation. For educational purposes it is beneficial to fully show everything.
- The goal of this thesis is not true machine independence, as we target only one architecture, but extensibility to other architectures should be considered.
- The back end should be a separate program, independent of the TinyC front end. This allows the TinyC front end used in the NI-GEN course to adapt to needs of the course without having to worry about our back end. And on the other hand it allows our implementation to define features which the TinyC front end doesn't want to have, but we need for runtime.
- The produced x86-64 assembly should be optimized not only for machine consumption by an assembler, but also for consumption by a *humans*—the assembly should be approachable by students.
- Since real programs rarely live in vacuum, the produced code should be able to interface with the operating system and foreign code.
- The back end targets real machine code, comparisons with other compilers will be possible and the relative performance of our solution should be evaluated.

Next subsections focus on the design and implementation of a TinyC back end for x86-64. Many of the algorithms used in the state of the art compilers have been introduced in chapter 3. This chapter focuses mainly on evaluating the algorithms and their practicality and possibility of use in our back end and highlights the chosen architecture of our back end (section 4.1). We also present data structures used for our middle end and back end representations separately (section 4.3) and only then delve into how we actually designed and implemented the individual components of our architecture.

4.1 Architecture

The input to our back end is a TinyC middle end IR, and the output is x86-64 machine code. This gives us our two main interfacing points. Everything in between is the goal of this thesis and the architecture should be designed to fulfill the goals presented above.

Traditionally compilers have been split into three main parts—front end, middle end, and back end (see section 3.1). Our compiler will have a bit of middle end (as it constitutes the input) and focus mainly on the back end. Often these three stages operate on completely different intermediate representations. In particular middle ends usually aim for representation suitable for (machine independent) optimization and often employs SSA form (see section 3.4). On the other hand, representations in middle ends usually aim for producing the best machine code possible, thus they focus on finding patterns in the compiled programs as well as the instructions sets, which allow choosing the most appropriate instruction sequence.

4.1.1 Middle end

Our compiler will need to process output of the canonical TinyC front end implementation[1], which is a middle end intermediate representation based on *control flow graphs* (described in section 3.2.4) and is in *value-based* SSA form (described in section 3.4.1).

We will follow this design decision. Not only to keep it approachable for students of the NI-GEN course familiar with the existing front end, but also because such representations are still considered state of the art even today.

Because our compiler will be separate program from the existing TinyC front end, we can choose how exactly to represent and even make a few design decisions that make it appropriate for our particular implementation and its needs. These design decisions are described more in later subsections (in particular 4.3.2 and 4.7).

4.1.2 Back end

Main interest of this thesis is the back end. As introduced in section 3.1 the main goal of a compiler back end is to translate from a (preferably machine independent) intermediate representation into machine instructions that can be executed by the processor.

For instruction selection, back ends usually use a specialized intermediate representation. For example, in the case of instruction selection by peephole optimization, these are the register transfer lists, or for tiling it can be trees, DAGs or general graphs.

While often register allocation is done on selected and ordered instructions, it doesn't have that many requirements on the actual IR. Though depending on the allocator, the IR has to to easily support calculation of *liveness*, which for iterative data-flow analysis

[1] `https://gitlab.fit.cvut.cz/NI-GEN/ni-gen-23`

requires the availability of control flow information, and the ability to iterate over *used* and *defined* registers.

In our back end, we are dealing with the x86-64 architecture, its machine constraints and calling conventions. Practical issues that limit us in our design are:

- Multi-output instructions like `idiv` (which produces both the quotient and the remainder) are not truthfully representable with trees and require DAGs.
- Some instructions like shifts require operands to be in certain registers.
- Not only are there two completely independent register classes (the general purpose registers and the "vector" registers), but both of them have aliasing subclasses which allow accessing only parts of the registers and operands of concrete instructions are limited to one concrete register subclass.

The register restrictions mean that register allocator has to either be able to represent *physical registers* in addition to *virtual registers*, or otherwise be able to restrict register assignment (like with *precoloring* in graph coloring register allocators, section 3.8.4.3).

Instruction selection is more decisive when it comes to intermediate representations. Advanced intermediate representations like DAGs and register transfer lists also come with additional problems—it has to be ensured that in every situation the nodes or transfer lists are translatable to machine instructions. This is a big problem for ensuring correctness.

The popularity of SSA for middle end optimizations and recent developments in the field of register allocation on SSA form make it very appealing even for a back end. However, methods for instruction selection on SSA are currently based on reduction to other problems. Similar reductions are possible for other components of a back end. Though for educational purposes, we want to focus on dealing with instruction selection and register allocation directly, and not focus on heuristics for solving NP-complete problems or offload the work to external dependencies.

For the very same educational reasons we also won't use something like `iburg` [Fraser et al., 1992b] to generate our instruction selector from a declarative description of the architecture.

Considering all these requirements, we decided to use *machine instructions* as the intermediate representation throughout the back end. This is based on several observations:

- We have to be able to represent machine instructions, since they are our output.
- Often a peephole optimizer step is run on the machine instructions in the final stage of a back end 3.3 and this is done over machine instructions.
- Instruction selection by peephole optimization (described in section 3.6.2) is a very competitive technique. It can also be used on a stream of machine instructions, though data-flow has to be used to achieve better quality.
- Correctness of instruction selection is more easily achieved if at all times the IR directly corresponds to machine instructions.
- Expressing machine constraints and calling conventions is straightforward in an IR based on actual machine instructions.
- Even though the machine instructions are *machine dependent*, their *form* and representation can be *machine independent*, still allowing for potential expansion to other targets.

If we use machine instructions for our back end, then full architecture of compiler can consist of the following steps:

- *SSA deconstruction.* Even though value-based SSA is convenient for the middle end, in the back end we have machine instructions, which work with *registers.* We also need to replace ϕ-functions with equivalent copies.
- *Code generation.* Translating from middle end to back end IR is often in general called *lowering*, because often we transform from a higher level representation to a lower level one. However, in our case, since back end IR represents the machine instructions directly, we can just call the phase *code generation*, since it will generate appropriate code for the target architecture. The code generator has to be machine specific, because it handles machine instructions, as well as calling conventions and other machine specific constraints.
- *Instruction selection.* Even though we produced valid instructions, the instructions are not necessarily the best possible ones. To improve the selection of instructions, we employ instruction selection by peephole optimization, inspired by Davidson and Fraser [Davidson et al., 1984a] (see also section 3.6.2).
- *Register allocation.* To simplify previous stages of our back end, they will mainly work with *virtual registers*, but they will also refer to *physical registers* to realize machine constraints and calling conventions. Before the machine code is finalized we have to rewrite the machine code so that it only refers to physical registers.
- *Peephole optimization.* Assignment of physical registers may unlock new possibilities for optimization. For example, due to coalescing (see section 3.8.3.4) copy instructions may become redundant, which can lead to other improvements, e.g. due to instructions now fitting into a peephole window. Because we use the same representation, the same peephole pass can be reused with a little care.

Notably we omit from our design instruction scheduling. As explained in section 3.7.3, modern implementations of architectures (including the x86-64) feature powerful out of order execution. Also, considering the ever increasing gap between speed of memory (and caches) and the processor itself, the chances of saturating execution units in a processor are only becoming smaller. Instead of specifically *optimizing* the instruction order for scheduling, we instead try to not introduce unnecessary *additional dependencies*, which is what makes any code scheduling (static or dynamic) more complicated, and the code ultimately slower on a superscalar processor. Notably this includes our use of the `setcc` instruction, where we avoid partial dependencies.

More complicated instruction selection mechanisms on specialized IRs have problems with ensuring correctness, because they may not be able to find a suitable instruction for a piece of the IR (subtree, register transfer list). In an extreme case the IR can be limited so that each piece of IR corresponds to one instruction. But, this has to be ensured across all targets, thus limitations in one target limit an IR common to all targets. Because we choose the initial instructions in the *code generation* step based on the middle end IR, we have to ensure that all our middle end IR nodes correspond directly to one or more machine instructions. This brings the limitations to the extreme—our middle end IR is essentially limited by all targeted architectures. This means that in the middle end IR we may have to limit ourselves to the most basic RISC-like operations, which are offered on all relevant architectures.

Limiting middle end, based on target architectures might seem like a bad thing, but we argue otherwise. A simple, regular RISC-like load-store architecture is easy to reason about in the middle end, and in fact may be preferable even if we had a free choice. Additionally, as seen in section 3.6.2, peephole optimization benefits from *expansion* into fundamental operations, because it can match them together into machine instructions based on the target architecture's available instructions and capabilities.

In our case, that role could be filled by a very naive code generator, which would generate the most simplest sequences of instructions and thus produce the “expanded IR” the peephole optimizer could then operate on. This makes the code generator simple, because not much case analysis is required. A smarter code generator employing more case analysis would even be undesirable, because it would hide low level patterns that could have been exploited by the peephole optimizer. This is inline with observations made in [Davidson et al., 1984b], where they also describe lack of case analysis in the lowering as a great benefit, and introduce the concept of an *intersection machine*, which implements only those operations available for all targets. Our middle end representation can thus be seen as an intersection machine.

4.1.2.1 Assembly

We have decided on our compiler to output textual assembly instead of machine code. While this means an external dependency, it is in our opinion a fair trade off, since producing actual machine code (i.e. executable binary files) is matter of serializing the machine instructions and adding headers depending on the executable format (like ELF, or PE).

The job of the back end is not made much easier by omission of serialization. As we also show in later sections, the compiler has to know exactly which instructions it intends to emit, otherwise it could easily use non-existing addressing modes. This distinction is in fact similar to the distinction of a *mnemonic* and an *opcode* (see section 2.2.1)—while assemblers expect the human friendly mnemonics, the compiler already has to figure out the right opcode.

The most useful job of the assembler is then the implementation of the serialization rules, which are sometimes tricky on x86-64. But more usefully, it also handles *relocations*, i.e. it adds notes for either the static or dynamic linker, which make the application work with external code. Emitting our own executables would mean that we could do without relocations altogether, but using external code is so interesting in its own right, that we prefer to keep the possibility (see also section 4.10.1).

Our assembly still has to keep in mind what kind of executable it produces. For example, we choose to output *positional independent code*, where we refer to procedures and global variables with *relative addresses*. Position independence is (usually) required for libraries, which get be mapped to different places in address space, and recently often also for executables (due to address space layout randomization). By making our code positional independent, we allow it to be linked into any external code.

4.1.2.2 Register allocation

The high level purpose of register allocation are simple—map virtual registers to physical registers. The details are much more complicated and explained in section 3.8. Here we focus on how to do register allocation in practice on our back end representation (machine instructions) for our target architecture (x86-64).

Since our back end representation is not in SSA form, we can’t use register allocation techniques requiring it. Though possibilities to use SSA form even for machine instructions exist (see section 3.4.2), in our opinion, the technique is little too disconnected from the traditional state of the art compiler structure, which deconstructs SSA before register allocation. For educational purposes we don’t think it would be a great idea to do register allocation on SSA form.

Local methods like top-down or bottom-up register allocation (section 3.8.4.1) are unsuitable, because their local nature just doesn’t allow them to produce great register assignment.

Linear scan, though a global algorithm, does also not quite fit our situation. As mention in section 3.8.4.2, the algorithm was designed for JIT compilers, which care very much about *run time* of the algorithm and for which more sophisticated methods already available at the time were too slow. We are designing an ahead of time compiler, which doesn't have to worry about compile time as much as JITs do.

Reduction of register allocation to another problem is interesting and produces good results, but as explained in section 4.1.2 we prefer to avoid reductions to another problems and instead like to stay close to the original register allocation problem.

Register allocation by graph coloring (see section 3.8.4.3 for more details) can also be seen as a reduction to another problem. What makes it different, is that usually implementations of graph coloring register allocation stay specialized—they don't just offload the work to an external graph coloring solver, but they code specialized solvers exploiting many particularities of register allocation like possibility of coalescing. Graph coloring register allocation also allows great flexibility with regards to machine constraints and also offers solutions for handling complex register class hierarchies. For this reason we think register allocation is the most suitable register allocation technique for our purpose.

There are many different improvements to Chaitin's original graph coloring register allocation. For example, *optimistic coloring* is a big improvement, while being very simple and efficient. Other improvements to Chaitin's algorithm mainly focus on improving coalescing and taking advantage of the special properties of interference graphs (see section 3.8.4.4).

Ultimately, we decided to implement iterated register coalescing as formulated in [George et al., 1996]. It has been regarded as the state of the art for a long time—newer algorithms take it as a reference, for example [Park et al., 2004] or [Pereira et al., 2005]. While optimistic coalescing seems great, it is not as straightforward extension as optimistic coloring. In fact, the exact approach is asymptotically impractical, and instead authors [Park et al., 2004] use heuristics instead. Graph coloring in perfect elimination order of [Pereira et al., 2005] is in today's eyes a remnant from time before special properties of SSA were understood. As a result it would be much preferable to use SSA-based register allocation due to [Hack et al., 2006], which also focuses on machine constraints and other problems encountered by practical register allocators.

Other advantage of iterated register coalescing is that, like some other formulations, but unlike for example [Pereira et al., 2005], it has an existing flexible and comprehensive method for handling register classes due to [Smith et al., 2004].

4.2 Technology

Our choice of technologies mainly concerns the programming language for the implementation and any other dependencies.

4.2.1 Assembler

For the assembler, we chose NASM[1]. It is an assembler intended for consumption of programs written by humans and because of this, the syntax is nice to read. Additionally, it is available on wide range of platforms, which makes it an easy dependency to obtain.

Other assemblers, like the GNU assembler, are not concerned much with human readability of the assembly, as they mainly consume assembly produced by compilers.

[1] https://www.nasm.us/

Unfortunately, NASM, like other assemblers, cannot be linked in to our compiler as a library, and we will have to either call it externally or leave it up to the users to assemble and link the programs.

4.2.2 Programming language

We considered the programming language by several criteria:

- Suitability for compiler development.
- Familiarity to the students of the NI-GEN course.
- How easy it is to compile and distribute the compiler.

Languages like OCaml or Scala are great for compilers, because they offer pattern matching and garbage collection, which both make compiler development easier. OCaml, unlike Scala is not taught in any class on the Faculty of Information and technology. Scala programs are also in our experience not as easy to distribute (due to dependencies on particular versions of either the Scala compiler or JDK).

We rejected emerging languages like Zig because they are still not stable enough and students are not familiar with them.

We deeply considered C++, because it is the language TinyC front end is written in. But, the language is improving much in each revision of the standard. Because of that, there is not a single C++ to target—there is anything ranging from C-like C++, to very modern C++20, which is sometimes not even fully supported, yet (especially by older versions of compilers present for example on stable Linux distributions).

Ultimately, we decided to implement the software in C. C programs can compiled and run basically anywhere. While the language doesn't support any data structure or paradigm well, it often means that we are able to choose the data structures most suitable to our exact problem. Hopefully everybody studying a TinyC compiler is already familiar with C.

Additionally, a back end written in C, can still be linked to and directly used by other projects involved in the Tinyverse.

4.3 Data structures

Presenting the data structures before presenting the implementation details and needs is a compromise. In practice in our implementation one motivated the other. Because the high level ideas of the used algorithms used in our architecture (section 4.1) are already explained in chapter 3, we hope that the design of data structures will be understandable enough even if it becomes fully justified only in later sections.

4.3.1 Middle end

The entire middle end is based on control flow graphs (CFGs) and the idea of *value-based SSA form* (presented in section 3.4.1). Most of the things a middle end works with can be represented as values. Including functions (represented by their addresses, i.e. they are *pointers to code*), blocks (also pointers to code), static variables (also represented by addresses, i.e. pointers to `.data` or `.bss` sections).

4.3.1.1 Values

Representing everything with values is great, because it is uniform and easy to work with. But there isn't a single kind of value—there are potentially very many if we count each kind of distinct operation as a separate kind of value. Different languages have different approaches for expressing the idea that a type has many different variants. Object oriented languages like C++ use *inheritance*. There, we would have a top level class `Value` and other classes like `Function`, `BasicBlock` or `Constant` that would inherit from it. Each subclass would hold data (*fields*) appropriate for the particular kind of value, but also fields inherited from the base class. Not only are different subclasses of different sizes, but code often wants to work with them opaquely—as such it only works with *references* to the values (either with real references or pointer as in C++ or through the classes being *reference types* in Java). Behavior supported for all values is defined in methods on the top level `Value` class, but subclasses can override it (or even may have to override it, if the method is abstract). Choice of the method is achieved with *dynamic dispatch* often implemented with *virtual tables*. Adding new value variants is easy and consists of introducing new subclass, which would override the methods from its parent class as appropriate. Adding new behavior to values is not that easy, since possibly all subclasses have to be touched and extended with method overrides to handle the new behavior.

Languages which support algebraic data types support a "type that has different variants" through *sum types*. With sum types, we can introduce a (top level) `Value` which would be a sum type of the types introduced for all the variants (like `Function` or `Constant`). Internally, this is usually represented either very similarly as the class hierarchy, with a separate representation for the different variants, each having only its fields. But there would be also one common field—the *discriminator*, a small integer distinguishing the variants. Though sometimes more efficient representation consists of a *tagged union*, where the sum type is big enough to store all the variants (like in a C `union`), but still contains the discriminating integer (the *tag*). Since the representation is uniform and the sum type has enough storage for all variants, it can be used to represent the type directly, there is no need for a level of indirection (so for example, array of `Value`s would be an array of structs not array of pointers to structs). Behavior would be implemented in functions, which would check the discriminator to choose the behavior for the variant at hand. Many languages aid the discriminator checking with more powerful *pattern matching*. Adding new behaviors is easy, since just one more function is added. Adding new variants is not as easy, because all functions have to be extended with the behavior for the new variant.

The class hierarchy and sum type approach are essentially complements each other—one makes it easy to add new variants and the other makes it easy to add new behavior. This is known as the *expression problem* [Wadler, 1998] and a lot of common languages (including C, C++, Java, Rust and OCaml) don't offer any easy solution. Since our implementation language is C, which can implement both approaches, we can decide on the approach freely and most appropriately to our use case.

Our `Value` type has a relatively *fixed set of variants*—there are arithmetic operations and constants like functions, basic blocks and global variable addresses. Extending with new variants is imaginable, but there isn't a large space for extensions here—there are just so many operations that a middle end can consider.

On the other hand, there are many different *behaviors* for middle end values. We want to print them, translate them to potentially many different architectures, and most importantly *optimize them*. There can be many different optimization passes over the middle end. The optimizations can be largely independent and benefit from being fully contained in a single place (either function or module).

For both of the above reasons sum types seem better for this use case. However, we can achieve the same with class hierarchy thanks to the *visitor pattern* described by [Gamma et al., 1995]. With visitor pattern we introduce a central dispatching place—the base `Visitor` class. This makes it hard to introduce new independent variants (the benefit of the class hierarchy approach), but allows new behaviours to be introduced by subclassing the `Visitor`. Different visitors can be introduced independently and also can group all the code implementing a single behavior, instead of interspersing it all over the subclasses constituting the variants.

However there is still a difference in the two and it is in how we work with the variants and how we *distinguish* between them. In a middle end optimizations often want to check for the nature of the values. For example, a peephole optimizer may want to check whether a value is an operation on two constants and replace it with a constant equal to the result (*constant folding*). Elsewhere we may need to check whether a load operation loads from a stack slot. This means that apart from checking the variant of a single value, we often want *nested checks*. Sum types with their discriminator allow such checks easily and languages with pattern matching allow even nice nested checks. In a class hierarchy for such questions we would have to either introduce methods for performing these checks (like `isConstant` or `isLoadFromStackSlot`) or employ some mechanism of investigating the variants, like with `dynamic_cast` in C++ or `instanceof` in Java (both of which are seen as very unidiomatic in their respective languages).

For our use of `Values`, sum types seem to be better match. But still, there are two options of implementation—*inheritance* (through struct embedding) and *tagged unions*. Ultimately we decided to go with inheritance for several reasons:

- Values are *recursive types*—values contain other values. For example an unconditional branch “contains” (has as a field) the destination of the jump. And this is true for essentially all operations which are just values produced from other values. Recursive types usually have to be implemented with a layer of indirection (in C this is done explicitly with *pointers*). Because of this, tagged unions which normally allow less indirection are not that beneficial.
- The sizes of individual values can differ greatly. For example there are usually not that many functions, but we may need to store a lot of information about them. On the other hand there may be many numeric constants, which essentially store just their tag and the constant itself. Tagged unions would have required us to have all the variants with the same size. Differences in size of the variants can be mitigated by adding a level of indirection to same variants and represent them with pointers. But having some variants with indirection and others not makes the structure slightly less nice to work with and since we need the level of indirection because of the recursivity, we may as well go with inheritance.
- Tagged unions require all the types used to represent the variants to be listed in the `union`. With inheritance new types can be introduced more independently, since most consumers of the types don’t care about the size of a “`Value`”, they work with pointers to values, i.e. “`Value *`”. Though the benefit of this independence is not big, since the tags still have to be listed in one central place (commonly an `enum` in C).

As of now the representation of `Value` looks like this:

```
typedef struct Value Value;
struct Value {
   ValueKind kind;
   u8 visited;
   u8 operand_cnt;
   Type *type;
```

```
    size_t index;
    Value *parent;
    Value *prev;
    Value *next;
};
```

`Value` is the base type other other types used to represent values inherit from. As such, it can have fields common to all variants, the obvious one being the discriminator, called `kind` in our case. In our implementation we found it useful to have also other fields:

- All values have a type (one of the TinyC types, e.g. "`int`" or "`char *`").
- Each value is given a unique index in its scope. For example, all operations in a function have distinct indices in the particular function. Each function has a distinct index from all other functions and basic blocks in one function also have distinct indices. Indices are assigned from zero and serve a few purposes:
  - Textual representation (see section 3.4).
  - Assignment to virtual registers (see section 4.5).
  - Indices to off the side temporary arrays containing information associated with values. Indices can essentially provide much faster and compact way of mapping values to something, then for example a hash table could.

  Since operations link to each other directly with pointers, the indices don't have much *semantic* meaning (except for the SSA deconstruction stage, where we violate that a little bit, see section 4.5). This for example means that the assignment of indices to values can be changed arbitrarily, without changing the meaning of values.
- Each value contains a link to its parent. For example, operations link to basic blocks they are contained in and basic blocks link to the functions they are in. This is can sometimes be convenient.
- Each `Value` has a link to next and previous one. This is mostly convenient for operations, which due to our control flow graph based representation have to be ordered. These links make up the explicit order. Since the linked list is contained in the structure itself, this is an *intrusive list*, (as opposed to for example something like a `std::list<T>` in C++, which is an *extrusive list*). Intrusive lists are a convenient way for representing linked lists in C.
- Even though not all values are operations, we still want to associate an operand count with each value. For values which are not operations, this count is simply zero. For some operations the operand count can be determined implicitly from the value's kind—for example multiplication has always two operands. But operations like phis or calls, which can have different number of operands (depending on the number of predecessor blocks or call arguments respectively) require the number of operands to be determined in other way than just with a check of the value's kind.

  We are able to determine the number of operands for ϕ nodes through parent link to a basic block, which stores the number of block predecessors. The number of operands of a call operation could be determined from the type of the first operand (the called function)—a function type also knows the number of arguments the function takes. However this became problematic when function with variable number of arguments became supported—for them the number of *parameters* is only lower bound for the number of parameters. For this reason we started storing the operand count explicitly.
- A `visited` field of values can be used by graph algorithms over values. This is used for example by the depth first traversal which computes the postorder of basic blocks in a function.

The choice of a linked lists instead of an array deserve a bit of attention. Operations in basic blocks are ordered through linked lists, because insertions, deletions and reordering in the middle end are convenient with them. Since our middle end is not complete and doesn't actually do that much, this amount of flexibility is not fully needed. But linked lists are convenient to work with in C, since C for example has no conveniences for dynamic arrays.

With ordinary doubly linked lists, there is *head* and often also a *tail* pointer which are like the entry points to the linked list—they are pointers from outside to the inside of the list. However, these bring several special cases to an doubly linked list implementation. For example a deletion of a first node in a linked list should (apart from changing the `prev` field of the second node) change also the head of the linked list. Even a construction of a list encounters a few special cases.

Circular doubly linked lists solve most of the problems. Since "last" node points to the "first" and vice versa, there is no longer a special case. All nodes can be treated as if they were in the middle. The problem with circular linked lists is that there can not easily be something like a `head` pointer from the outside to the linked list—or rather there could be, but it would bring back the special cases of non-circular doubly linked list. But in our use case, we want to link together operations inside a basic block to a linked list rooted in the basic block—a basic block should contain a pointer to the head (or also the tail) of the linked list of instructions. Our solution is to make basic blocks part of the doubly linked list of instructions they contain. This can be done, since basic blocks inherit from `Value` as well and thus have `next` and `prev` fields and can be part of the list. Then `next` and `prev` fields of the basic block serve as the `head` and `tail` pointers to the list of instructions. But thanks to the linked list being circular, they are updated transparently.

Because the `next` and `prev` fields of basic blocks are used for holding the instruction lists, they may not be used to link together all basic blocks in a function, thus we need to handle it differently (discussed in section 4.3.1.5).

4.3.1.2 Constants

Here we show how some concrete value variants are represented:

```
typedef struct {
   Value base;
   i64 k;
} Constant;

typedef struct {
   Value base;
   Str name;
   Value *init;
} Global;

typedef struct {
   Value base;
   Value *operands[];
} Operation;
```

All values inherit `Value` through *struct embedding*. For consistency, all value variants inherit `Value` in a first field called `base`. This makes it easy to both downcast (`(Global *) value`) or upcast (`&global->base`), since for example pointer to `Constant` and its `Value` base are the same.

Both Constant (which represents actual integer or character constants) and Global represent constant values. Their fields should identify them sufficiently. For numeric constants, this can be done with a single field holding a 64-bit integer (the largest size supported by TinyC). Global values are more interesting as they are named. With external linkage, we need to preserve their names. Also, unlike normal variables allocated on the stack, global values are special, because they may only be initialized with *constants*. For simplicity, we store the initializer with the Global. If there is no initializer (init is NULL), then the global variable is meant to be zero initialized (i.e. it will be put into .bss section and the initializer doesn't have to be stored in the resulting binary). Allowing only constant initializers is the same behavior as in the C programming language.[1] Type of a Global is actually a *pointer* to the type of the global variable—in a sense the value represents the address of the global variable, not the global variable itself.

4.3.1.3 Operations

Operations are the main kinds of values which reference other values. Operations consist purely of the base and an array of operands. The array of operands uses a feature of the C programming language called *flexible array member*, which allows the array to be part of the structure, but to be dynamically sized. This way, the same Operation representation can work with arbitrarily many operands—though the structure has to be dynamically allocated with the correct size. The actual number of operands is given by the operand_cnt field of the base, as explained above.

This representation of operations is very uniform and allows tasks as iteration over all operands very easily. This simplifies operations on the middle end representation quite a lot and requires no case analysis, since values that don't have operands simply have operand_cnt of zero. Iteration over all operands is very common for many operations on the middle end IR—printing, translation to machine code, use analysis, etc.

On the other hand, consider how much case analysis and special provisions for iteration would have to be made if for example unary and binary operations were represented like this:

```
typedef struct {
   Value base;
   Value *arg;
} Unary;

typedef struct {
   Value base;
   Value *left;
   Value *right;
} Binary;
```

Though the advantage of this representation is, that operands are available through human readable names like left and right. For our operation structure, this can be remedied through macros:

```
#define OPER(v, i) (((Operation *) (v))->operands[i])
```

[1] If a global variable were to be initialized with a non-constant value it would be questionable *when* the value should actually be evaluated—for example the global variables should certainly be initialized before main runs, but in what order, and how to run code before main? C++ allows non-constant initializers and runs them through hooks which execute before main and which are offered by the C runtime which is the one calling main.

```
#define UNARY_ARG(v)     OPER(v, 0)
#define BINARY_LEFT(v)   OPER(v, 0)
#define BINARY_RIGHT(v)  OPER(v, 1)
```

Here the mapping of indices to binary or unary operation operands is reasonably clear. But the macros are the are much useful in cases where there the order of operands is not so obvious. These are mainly the operations which don't correspond to classic arithmetic, and which we introduce here:

- *Load.* The load operation has one operand, the address for the load, while the load operation itself represents the loaded value.

  ```
  #define LOAD_ADDR(v) OPER(v, 0)
  ```

- *Store.* Store operation has two operands, one is an address (target of the store) and the other is the value that is to be stored. Unlike most other operations, store doesn't evaluate to anything—it is only useful for its *side-effect* on memory.

  ```
  #define STORE_ADDR(v)  OPER(v, 0)
  #define STORE_VALUE(v) OPER(v, 1)
  ```

  Currently, we model the fact that an operation doesn't return anything with the TinyC `void` type. This is not entirely correct, since the store operation *itself* represents a value, while `void` represents an *absence* of a value altogether. A bit more flexibility could be achieved if the stores would be of the *unit type*, but so far we kept the type system used in the middle IR to exactly the same as the one used for the TinyC programming language.
- *Call.* Call operation has as its first operand the function to be called (a value with type of function pointer), and the arguments are in the operands starting at index one. The call operation evaluates to the return value of the function and has its type.

  ```
  #define CALL_FUN(v)  OPER(v, 0)
  #define CALL_ARGS(v) (&OPER(v, 1))
  ```

- *Jump.* Jump operations perform unconditional jump to a basic block. The basic block is the only operand of the operation.
- *Branch.* Branch operations perform conditional jump to one of two basic blocks, based on a condition. The condition is stored as the first argument, the basic block which is the destination in case the condition evaluates to true (non-zero) is the second argument, and the destination in case the condition evaluates to false (zero) is the third argument.

  ```
  #define BRANCH_COND(v)  OPER(v, 0)
  #define BRANCH_TRUE(v)  OPER(v, 1)
  #define BRANCH_FALSE(v) OPER(v, 2)
  ```

- *Return.* There are actually two kinds of returns—those that return a value (i.e. `operand_cnt` is 1) and those that don't return a value (i.e. `operand_cnt` is 0). Absence of a return value could also be represented with a return of a unit value of the unit type, but this was again rejected to stay in line with the TinyC type system.

Currently, the type of the control flow changing (*terminator*) instructions is the TinyC `void` type. Some languages employ a *bottom type* (sometimes also called *zero type* or *never type*), which has no values and thus can be more suitable for representing control flow changing operations, since they don't ever return.

4.3.1.4 Basic blocks

Basic blocks apart from being the heads of the linked lists of instructions through their `next` and `prev` pointers in `base` also explicitly hold the *predecessors* through a dynamically growable array:

```
typedef struct {
   Value base;
   MBlock *mblock; // link to corresponding machine block
   Block **preds_;
   size_t pred_cnt_;
   size_t pred_cap_;
   size_t depth; // loop nesting depth (0 means outside of all loops)
} Block;
```

In TinyC IR blocks can gain *successors* only through jump and branch terminating operations. Thus block's successors can be found implicitly by investigating it's last operation through the `base.prev` field. Explicitly storing successor blocks would only lead to duplication, which can easily result in inconsistencies. One special value held by basic blocks is the loop nesting depth which is used to calculate spill costs (see section 4.9.7.1). There is also a link to a machine block, which is further explained in section 4.3.2.

Other operation that deserves mention is the ϕ, denoted with `phi` in the textual form of the IR and often simply written as "phi". They are deeply connected to blocks—while still ordinary operations, their operands correspond to values from predecessors. In our representation this correspondence is implicit: ith operand of a phi operation is the value from the ith predecessor. This needlessly requires care from any code tries to change control flow, and may encounter phi operations. However such code will always require special care, because of the non-standard nature of phis.

4.3.1.5 Functions

Representation of functions is show below:

```
typedef struct {
   Value base;
   size_t index;
} Argument;

typedef struct {
   Value base;
   Str name;
   Argument *args;
   Block *entry;
   Block **blocks;
   Block **post_order;
   size_t block_cap;
   size_t block_cnt;
   size_t value_cnt;
   MFunction *mfunction; // link to corresponding machine function
} Function;
```

As mention in a previous section, we usually don't want to iterate over the blocks in some random order or the order they were created in. Instead, we want to iterate over them in an order that is beneficial for performed analysis and optimization passes. Often this order is either reverse postorder (where all non-cyclic predecessors are visited *before* the block itself) or postorder (where a block is visited *after* all its non-cyclic predecessors), which is useful for example for liveness analysis (section 4.9.1).

Postorder is easily computed with depth first search [Aho et al., 2006], and we store it as an array of (pointers to) basic blocks. The array is dynamically growable, since blocks can be added or deleted to functions. The only explicitly stored basic block for a functions is the *entry* basic block. All useful blocks are (recursively) reachable through it and listed in the postorder. By using the postorder to iterate over the blocks, unreachable blocks are automatically skipped. Iteration over an array with the postorder can be also done in reverse, so in a way a reverse postorder is simultaneously also available.

Special values are function arguments. From the perspective of the called function they are constants. Because of this, they don't appear in the control flow graph in basic blocks—they are not operations that run. Through pointers, other values like operations can link to them freely, but since sometimes iteration over all arguments is needed, they are also explicitly linked from the function itself. The number of function arguments doesn't have to be stored explicitly—it is derivable from the `type` stored in `base`. Arguments could be linked in an intrusive linked list, but storing them in an array and explicitly storing the index of each argument proved to be convenient.

Function with variable number of arguments are currently not allowed to be defined in TinyC, but they wouldn't require many special provision here—handling of variable arguments such as with C's `va_arg` is stateful (arguments are extracted one at a time) and thus a corresponding `va_arg` would have to be an operation over a `va_list`, thus no longer a constant, and very distinct from `Argument`.

4.3.2 Back end

This section describes the reasoning, motivation and design of the data structures core to the backend. There are three main data structures in the back end which are introduced in the following subsections:

1. Machine instruction (section 4.3.2.2).
2. Machine block holding a contiguous sequence of machine instructions (section 4.3.2.3).
3. Machine function holding a linear sequence of basic blocks.

4.3.2.1 Registers

Before introducing actual data structures, we note how we represent *registers*. There are two kinds of registers: *virtual registers* and *physical registers*. Not all compilers need to represent physical registers, but we need to, since we use the same representation throughout the back end, and use physical registers to express *machine constraints* (see section 3.8.3.6) even before register allocation maps virtual registers to physical registers.

Representing registers with integers, instead of some kind of rich object, has many advantages:

- They are small and easy to understand.
- Data can be associated for each register with an array indexed by the registers. This is compact, easy from memory management stand point of view, and provides great cache locality from the point of view of associated data. Additionally, many different kinds of data may be associated with the registers at different times.
- Physical and virtual registers can be assigned different ranges of integers and thus be distinguishable just by the numbers themselves.

- Associating physical registers with low numbers can allow them to be used as bit positions in bit sets packed into a 64-bit integer or similar, allowing register sets or masks to be efficiently and concisely represented.

For these reasons we chose to represent integers with an integer type. Apart from physical registers and virtual registers we have found need to represent "no register". A special integer value is suitable for this. In the end we decided on the following:

1. Special register value `R_NONE` is assigned the integer 0.
2. Physical registers are named symbolically through an enum as `R_`⟨*name*⟩ and are assigned the indices 1 through the number of physical registers.
3. Virtual registers are all integers larger than the last physical register. They are denoted with `t` and their number. For example on x86-64 our first virtual register is `t17` (since there are 16 physical registers). But we will take the liberty of using any virtual registers in examples, even though on the x86-64 they might actually clash with physical registers.

We considered representing virtual registers by the highest bit (or similarly with negative numbers), but this doesn't allow all registers to be used as indices. We found representing the special register value with zero is sometimes convenient and sometimes not—zero makes zero initialization automatically assign "no registers" and is convenient in checks, but it shifts all physical registers by one, which makes the assignment not consistent with machine encoding.

4.3.2.2 Machine instructions

Machine instructions usually consist of two essential parts:

1. *Opcode.* An opcode usually packs together a few things:
   - the operation itself,
   - the number of operands and their kinds.
2. *Operands.* Operand kinds are usually:
   - registers,
   - immediates,
   - and memory locations.

One possible representation for instructions contains the opcode and the operands, where the operands also encode their kind. In such representations, tagged unions can be used with advantage: the `kind` field in a structure encodes the operand kind, while the (anonymous) `union` allows the storage for the data ("payloads") of all the different kinds of operands. This representation of x86-64 instructions is sketched here:

```
typedef u32 Register;
typedef u64 Immediate;

typedef enum {
   ML_REG,
   ML_REG_DISP,
   ML_BASE_INDEX_DISP,
   ML_RIP_DISP,
   [...]
} MemoryLocationKind;

typedef struct {
   MemoryLocationKind kind;
```

```
    union {
        struct {
            Register reg;
        } reg;
        struct {
            Register reg;
            Immediate displacement;
        } reg_disp;
        struct {
            Register base;
            Register index;
            Immediate displacement;
        } base_index_disp;
        struct {
            Immediate displacement;
        } rip;
        [...]
    };
} MemoryLocation;

typedef enum {
    O_REG,
    O_IMM,
    O_MEM,
} OperandKind;

typedef struct {
    OperandKind kind;
    union {
        Register reg;
        Immediate imm;
        MemoryLocation mem;
    };
} Operand;

typedef enum {
    [...]
} OpCode;

typedef struct {
    OpCode opcode;
    Operand operands[];
} Instruction;
```

Both registers (see section 4.3.2.1 for more details) and immediates are represented directly with integer types.

Array of operands is represented with a flexible array member, which means that number of allocated operands may be customized for each instruction. On x86-64, instructions have at most three operands, so the array could be statically sized, if wasting a bit of memory each instruction is acceptable.

The number of operands can be usually derived from the opcode. But this implies, that two and three operand multiplications have to be distinguished by the *opcode*, even though they use the same *mnemonic* in assembly. See section 2.2.1 for more details.

One problem with the representation is, that it allows invalid forms of instructions to be easily represented. For example more than one memory operand is allowed, even though x86-64 only allows at most one. Since `MemoryLocation` is the biggest member of the `union`, this also makes the struct larger than necessary if only one memory operand is ever used.

The representation of x86-64 memory locations poses a bit of a problem, since there are many ways of specifying a memory location (see section 2.2.3). In the usual SIB mode essentially all components (the base register, index register, scale and displacement) are optional. Having a variant for each combination soon becomes unwieldy—not just to list the variants, but to work with them. For example a function checking whether the memory location uses an index register would have to check whether the kind of the instruction is one of the ones that use an index register. Iteration over all involved registers is also a bit cumbersome, because the registers are mixed with immediates and different variants of the union store registers differently.

Instead a more flat representation, where all the fields are in the struct (without any unions) can simplify matters, because it is more uniform:

```
typedef struct {
   MemoryLocationKind kind;
   Register base;
   Register index;
   Immediate scale;
   Immediate displacement;
} MemoryLocation;
```

This representation is however also problematic, because the `union` essentially became implicit. Not all fields are valid, and this depends on the kind, which still has to be investigated. Even iteration didn't become much easier because of that. However, the differences between the memory location variants are mostly due to fields being optional.

Instead of deriving the validity of the fields from the `kind`, we could reserve one special value for each field to mean that the field is actually not present. For displacements this is very natural, since absence of a displacement is very similar to the displacement being zero.[1] For scale, similar thing applies—absence of scale and scale being one are essentially equivalent. For registers a special value would have to be reserved meaning that the register is not used. Normally, the compiler would have to be very careful not to end up with an invalid combination of absent fields, but since with x86-64 memory locations *all* fields are optional, it is not even a problem. The representation could thus be simplified to:

```
typedef struct {
   Register base;
   Register index;
   Immediate scale;
   Immediate displacement;
} MemoryLocation;
```

Iteration over the registers is a bit easier—as long as the invalid register value is handled specially everywhere, it is possible to iterate over both `base` and `index` un-

[1] It is very different in the actual *encoding* of the instruction, since modes with displacement need the displacement to be present even if it is zero. But this is a detail that can be handled by a future stage of the compiler. Just like the fact that on the x86-64 the displacement can be either 8 bit or 32 bit. Choosing the 8 bit form may be deferred to much later stage of the compilation. This is different with 64 bit displacements. Since no instruction allows 64 bit displacement, the late stage of compilation shouldn't ever need to encode such displacement, this should be handled in instruction selection phase, because different *instructions* have to be used.

conditionally. Though still it is not ideal, because the iterator would either have to be callback based or would have to employ code duplication to work on both of the register fields. Arrays are much nicer for iteration, which leads to the following idea:

```
typedef struct {
   Register reg[2];
   Immediate imm[2];
} MemoryLocation;
```

Arrays are ideal for iteration. Though the problem is now accessing specific register or immediate, because hardcoded indices have to be used to access them in the arrays. Some kind of nested anonymous `struct` and `union` combination would have to be used to make both possible:

```
typedef struct {
   union {
      struct {
         Register reg[2];
         Immediate imm[2];
      };
      struct {
         Register base;
         Register index;
         Immediate scale;
         Immediate displacement;
      };
   };
} MemoryLocation;
```

Or macros could be used to abstract away the ugly indexing:

```
typedef struct {
   Register reg[2];
   Immediate imm[2];
} MemoryLocation;

#define BASE(mem)         ((mem)->reg[0])
#define INDEX(mem)        ((mem)->reg[1])
#define SCALE(mem)        ((mem)->imm[0])
#define DISPLACEMENT(mem) ((mem)->imm[1])
```

With either, we have essentially *flattened* the memory location representation to a form that allows easy addressing of individual fields as well as iteration.

Going back to the full instruction representation, we can apply similar flattening principles. In particular, we can also make the representation even more uniform, by noticing that even memory locations, the most structured kind of operand, are made up of just registers and immediates. And registers as well as immediates can be represented by integers. Now the memory locations, immediates and registers can all fit into one uniform structure containing an array of integer “operands”:

```
typedef long Operand;

typedef struct {
   OpCode opcode;
   Operand operands[];
} Instruction;
```

The meaning of individual slots in the operands array could either depend on the opcode, or be the same for all opcodes if we are willing to sacrifice memory and keep representation of all instruction uniform—in that case the flexible array member could also be replaced by fixed size array.

In the end, the following representation is what we chose for our implementation:

```
typedef u32 Oper;

typedef struct Inst Inst;
struct Inst {
   Inst *next;
   Inst *prev;
   u8 kind;
   u8 subkind;
   u8 mode;
   [...]
   Oper ops[];
};
```

The split of opcode into `kind` and `subkind` is beneficial, because there are opcodes that have very similar characteristics and handling them all at once through the `kind` field is convenient. For example while there are 16 different conditional jump opcodes, most of the time we don't care much about the particular opcode (`subkind`), just the fact that it is an indirect jump (`kind`). Other groups form nicely:

- binary ALU operations (`add`, `sub`, `xor`, `and`, `or`, `test`, `cmp`, `imul`),
- unary ALU operations (`not`, `neg`),
- shifts (`shl`, `sar`, ...),
- conditional moves (`cmovz`, `cmovl`, . . .),
- conditional jumps (`jz`, `jl`, . . .),
- set on condition (`setz`, `setl`, . . .),
- long division and multiplication (`idiv`, `imul`),
- etc.

Conveniently conditional jumps, conditional moves, and conditional set instructions are all based on the same 16 *condition codes* (see section 2.2.5), so `subkind` can be a condition code for all three of these.

The `mode` field is what gives the meaning to the individual slots of `ops` (operands). In theory, modes could just differentiate between the kinds of operands used. For example, two operand instructions usually have the following modes:

- register, register
- register, memory
- memory, register
- memory, immediate

However, for register allocation we want more information about the involved registers. In particular, we want to iterate over all the *defined* and all *used* registers separately. Registers used for forming memory locations are only used, never defined, even if the instruction is store instruction which writes a value—the value is stored to *memory*, not to the register. But other register operands may actually be either both used and defined, or one of those. For example, consider the following instructions:

```
mov rax, 1        ; rax just used
add rax, 2        ; rax used and then defined
```

```
imul rbx, rax, 4 ; rbx just defined, rax just used
cmp rax, rbx     ; rax just used, rbx just used
mov [rax], rcx   ; rax just used, rcx just used
mov rax, [rcx]   ; rax just defined, rcx just used
xor [rbp+16], -1 ; rbp just used
```

Notably, `add` and `cmp`, which on the x86-64 are very similar and can be grouped under the same `kind`, have a different mode even in the two register form: `cmp` unlike `add` doesn't write to the first register. The ability to tell this just from the `mode` alone without investigating `kind` is one of the benefits of splitting them, instead of having a single `opcode` field.

`kind`, `subkind` and `mode` are all small integers. And while they have their meaning by themselves, other information may be associated with them by considering them as indices into arrays with associated information for each. Such arrays can be used to hold for example string representations of kinds and subkinds.

These "descriptor arrays" are more interesting for modes. They list for each mode which part of the operands correspond to *defined* and which to *used* registers:

```
typedef struct {
   u8 def_start;
   u8 def_end;
   u8 use_start;
   u8 use_end;
   [...]
} ModeDescriptor;
```

The `def_start` field gives the starting index of *defined* registers in `ops`, `def_end` the index one past the last defined register. Analogously for *used* registers. In practice, mode descriptors can look like this:

```
ModeDescriptor formats[] = {
   // first only defined, first two used,
   // e.g. add rax, rcx
   [M_Rr] = { 0, 1, 0, 2, },
   // none defined, first two used
   // e.g. test rax, rcx
   [M_rr] = { 0, 0, 0, 2, },
   // first defined, second used
   // e.g. mov rax, rcx
   [M_Cr] = { 0, 1, 1, 2, },
   // first defined, first three used
   // e.g. add rax, [rcx+rdx]
   [M_RM] = { 0, 1, 0, 3, },
}
```

The naming convention for modes has one operand per each letter after underscore. Here the letter `r` means only used register, `R` used and defined register, `C` only defined register and `M` stands for memory. This representation with indices is nice for x86-64, because it allows *overlap* between the used and defined registers. For example, the first mode uses and later defines the same register, hence it is covered by both ranges.

The reason for representing modes with small integers as indices into arrays, as opposed to, for example, pointers to the descriptors (which would allow even more opaqueness and similar target independence of the representation), is not only because the integers can be much smaller than pointer, but ultimately in our peephole optimizer we want to do *pattern matching* over the instructions. Small integers are more flexible in this regard: for example matching one of multiple modes can be done with bitwise operations instead of sequential comparisons of pointers. The integers are also more flexible since more tables can be associated with the same indices (or the same table can be used for subkinds of different kinds of instructions, like with the condition codes for `cmovcc` and `jcc` x86-64 instructions, which proved to be useful and elegant in the implementaiton).

One of the important aspects of the representation is, that each register logically present in an instruction is present *only once* in the representation. Alternatively a different representation could have forbidden any overlap between used and defined registers, and if a register is both used and defined, then it would be listed twice in the representation. However, since there is only one register logically, only one of these two would get serialized (e.g. or printed to assembly) and both of them would have to be kept in sync. Keeping the two mentions of the same registers in sync is tricky in situations like spilling, where uses and definitions of a register are replaced by loads and stores through fresh virtual registers. In case a register that is both used and defined, a single virtual register has to be used for both the load and store.

The layout of the `ops` used for x86-64 is illustrated in figure 4.1. The representation fits operands for each mode into just 6 operand slots. Not all modes use all 6 slots, but they are allocated with them nonetheless, since it makes it universally possible to just change mode and some operands to transform one instruction into another. It also simplifies memory management, because for objects of single size, memory fragmentation is not a problem.

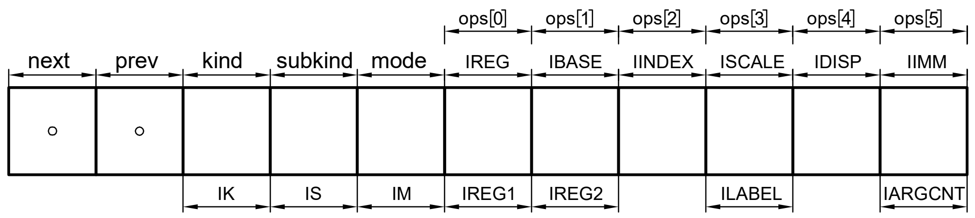

Figure 4.1. x86-64 instruction representation

All 6 operand slots are needed for only one instruction: 3 operand `imul` with register (1 slot), memory (2 register and 2 immediate slots) and immediate (1 slot) arguments.

Different slots are used for different purposes in different addressing modes. But the assignment has been kept consistent as long as possible. For example, most modes have one main register, which is always in the first slot. The single memory operand always uses the slots 1 through 4. Slot 1 is also used in case there are two registers involved—in which case memory operand is not used. The following macros are used to access the operands:

```
#define IREG(inst)     ((inst)->ops[0])
#define IREG1(inst)    ((inst)->ops[0])
#define IBASE(inst)    ((inst)->ops[1])
#define IREG2(inst)    ((inst)->ops[1])
#define IINDEX(inst)   ((inst)->ops[2])
#define ILABEL(inst)   ((inst)->ops[3])
#define ISCALE(inst)   ((inst)->ops[3])
#define IDISP(inst)    ((inst)->ops[4])
#define IIMM(inst)     ((inst)->ops[5])
#define IARG_CNT(inst) ((inst)->ops[5])
```

The fact that different slots have different purposes in different modes sadly makes the representation a bit harder to understand. On the other hand, most of it is hidden behind accessor macros and the really close to the encoding of the x86-64 instructions. Of course the actual x86-64 encoding is much more compact. But for example the following two instruction (load and store) are encoded in the same way in both the backend representation and the actual x86-64 encoding:

```
mov rax, [rdx+2*rcx]
mov [rdx+2*rcx], rdx
```

In x86-64 serialization, the only difference is one bit in the opcode field (the *direction bit*), while in our backend representation they have different mode (`M_CM`, i.e. register and memory vs `M_Mr`, i.e. memory and register).

So far, the fact that `Oper` is defined as 32-bit unsigned integer has been neglected. It may seem as a weird choice, considering that immediates are usually considered signed by the architecture. In practice, there are two good reasons for this:

- Immediates for most x86-64 instructions are limited to 32 bits. So limiting them to the same range makes it obvious that special handling of larger immediates is needed.
- Unsigned integers have defined representation and behaviour in C, as well as on the x86-64. With signed integers this is not true: C doesn't define the representation of signed numbers, while on the x86-64 signed integers use two's complement representation. When, for example, evaluating constant expressions in the peephole optimizer, the semantics of the x86-64 should be used, *not* the semantics of machine the compiler is running on. Two's complement semantics should be used for immediates, which is easier to do portably on unsigned numbers.

Notably the "move immediate into register" (i.e. `mov rax, 1`) is the only x86-64 instruction supported by our backend, which allows a 64-bit immediate. It stores the immediate in two 32-bit operand slots. The large immediate is only accessible through two accessor functions, which signalize the need for careful handling.

The fields of the `Inst` structure are usually accessed through macros:

```
#define IK(inst) ((inst)->kind)
#define IS(inst) ((inst)->subkind)
#define IM(inst) ((inst)->mode)
```

Naming is intentionally very terse, because especially in the peephole optimizer, they are used *a lot*—in our opinion long names would bring no sizeable benefits. The "`I`" prefix on all the macros stands for "`Inst`", which accomplishes at least some namespacing in C.

Another so far neglected aspect of the modes and operands were *labels*. Even though ultimately all instructions compile out the memory location displacements into up to 32-bit numbers, some addresses are *logically* connected to (often named) objects. In the following TinyC example, the address of the second field of a global variable has an address relative to the start of the global variable:

```
struct S {
    int a;
    int b;
};

S global_struct;

[...]
```

```
    global_struct.b = 5;
[...]
```

We want to (or even need to) use RIP-relative addressing for global variables (see also section 2.2.3. RIP-relative addressing uses the instruction pointer (`rip`) and a 32-bit displacement. However, the real displacement to `global_struct.b` or even to `global_struct` is not known until after the final executable is *linked* together. As a zero-initialized global variable, `global_struct` will be put into the `.bss` section. During linking, `.bss` sections from all object files are merged into one `.bss` section, so the real displacements cannot be known until all object files are linked together. Hence earlier compiler stages need to somehow encode relative address without knowing the final displacements. Without going into further details, in ELF object files this is done through relocations. Assemblers usually support labels, for example our target NASM would allow the following:

```
mov qword [rel global_struct + 8], 5
```

The `rel` keyword is important, as it forces RIP-relative addressing. The assembler doesn't write out the "label plus offset" information into the object file, it resolves the address to a relative position in this object file's (in this case) `.bss` section. Our compiler has to use the NASM syntax with the label. But even if it did produce object files directly, it would be necessary to associate the address of global variable's field with the global variable itself.

For this we use the concept of *labels* are used. Like other `Oper`s, labels are just integers. In this case each integer corresponds to an entry in a label array, which has a pointer to a `Value`. Hence a label can freely point to global variable, function, strings literal, etc. But importantly the label operand is stored separately from the displacement operand—in a distinct operand slot. So there is still a displacement involved, but only relative to the label. Since labels are only interesting in RIP-relative addressing, which doesn't use neither base, nor index register or scale, their slots can actually be reused—label is stored inside scale's slot and special register value in base's slot is used to signalize RIP-relative addressing. Displacement is stored in the same way in both addressing modes. We currently use `R_NONE` as the special value for RIP-relative addressing. This is convenient, because it is already a special case in a lot of places in code. Slight problem is that this makes memory locations without the base unencodable in our representation, but in practice we haven't yet found much need to have memory locations without the base other than the RIP-relative addressing.

Instructions also contain pointers to the previous and next one. Like `Value` in the middle-end, they are linked together in an *intrusive linked list.* Linked lists allow arbitrary insertions, deletions and reorderings in the middle of a sequence of instructions, which are exactly the operation a peephole optimizer does. Circular doubly linked list is especially nice for these operations, because it has no special cases for insertion or deletions at the end or the beginning of the linked list, simply because there is no real start or end of the linked—every node may be presumed to be in the middle.

4.3.2.3 Machine blocks

To allow circular linked lists for instructions to work nicely the *head* of the linked list (i.e. the pointer to the linked list of instructions) should also be part of the linked list itself. With our middle-end representation this was fairly easy, since "instrucions" were `Value`s with next and previous links, and basic blocks were values as well, so the next and previous links of basic blocks served the purpose of the "head" and "tail" fields.

Similar thing would be needed here to allow machine basic block to be the head of the linked list, but also *part* of the linked list of instructions. This can be achieved by "inheriting" the `Inst` struct in machine basic block struct:

```
struct MBlock {
   Block *block;
   size_t index;
   // `insts.next` and `insts.prev` are respectively the head and tail of
   // circular doubly linked list of instructions
   Inst insts;
};
```

Adding `Inst` as field of `MBlock` means that it is now able to be part of the linked list of instructions. The next and prev pointers serve as head and tail respectively. Other fields like `mode` or `kind` are also inherited—special values can be reserved for them and for example peephole optimizer's pattern matching on instructions can then transparently skip machine basic blocks, if it gets to them while investigating neighbours of instructions. As a result of the representation, the rolling window in the peephole optimizer actually transparently skips patterns that would reach out of bounds, because basic block's kind doesn't match any usual pattern.

Other than that, currently a machine block just links back to a middle-end basic block. Since currently the backend lacks the ability to do big changes to control flow, it can reuse the control flow (successors and predecessors) of the middle-end representation.

Figure 4.2 shows examples of a few x86-64 as represented in a linked list of `Inst` structures together with the head of the linked list an `Inst` in `MBlock`.

4.3.3 Work lists

Often we found ourselves in need of a data structure like a queue or stack, which would additionally allow checks of *presence* of elements in the queue or stack. Such data structure is called a work list and is used for example for liveness data-flow analysis (see section 4.9.1).

A right set implementation is suitable for a work list. Especially in our case, where our elements are mostly small integers (indices). A *bit set*, is very compact, but it doesn't support pushes or pops of elements efficiently.

In the end, we extended Briggs' [Briggs et al., 1993] sparse set data structure into a circular buffer. The result is a set, which allows fast presence checks, is fast to iterate over (the elements are stored contiguously in an array) and to which we can either prepend or append freely.

This one data structure covers almost all our needs for various different stacks, queues and work lists. The only downside of the data structure is its memory inefficiency, so we notably don't use it to represent interference graphs (see section 4.9.5).

4.4 Critical edge splitting

Critical edges and their implications for SSA deconstruction are explained in section 3.5. We don't have to split *all* critical edges. From the perspective of our back end, only edges to blocks with ϕ functions are problematic (see also section 4.5). But we choose to split all critical edges, since later optimizations can undo the splits, if it turns out they are not needed after all (see section 4.8.4).

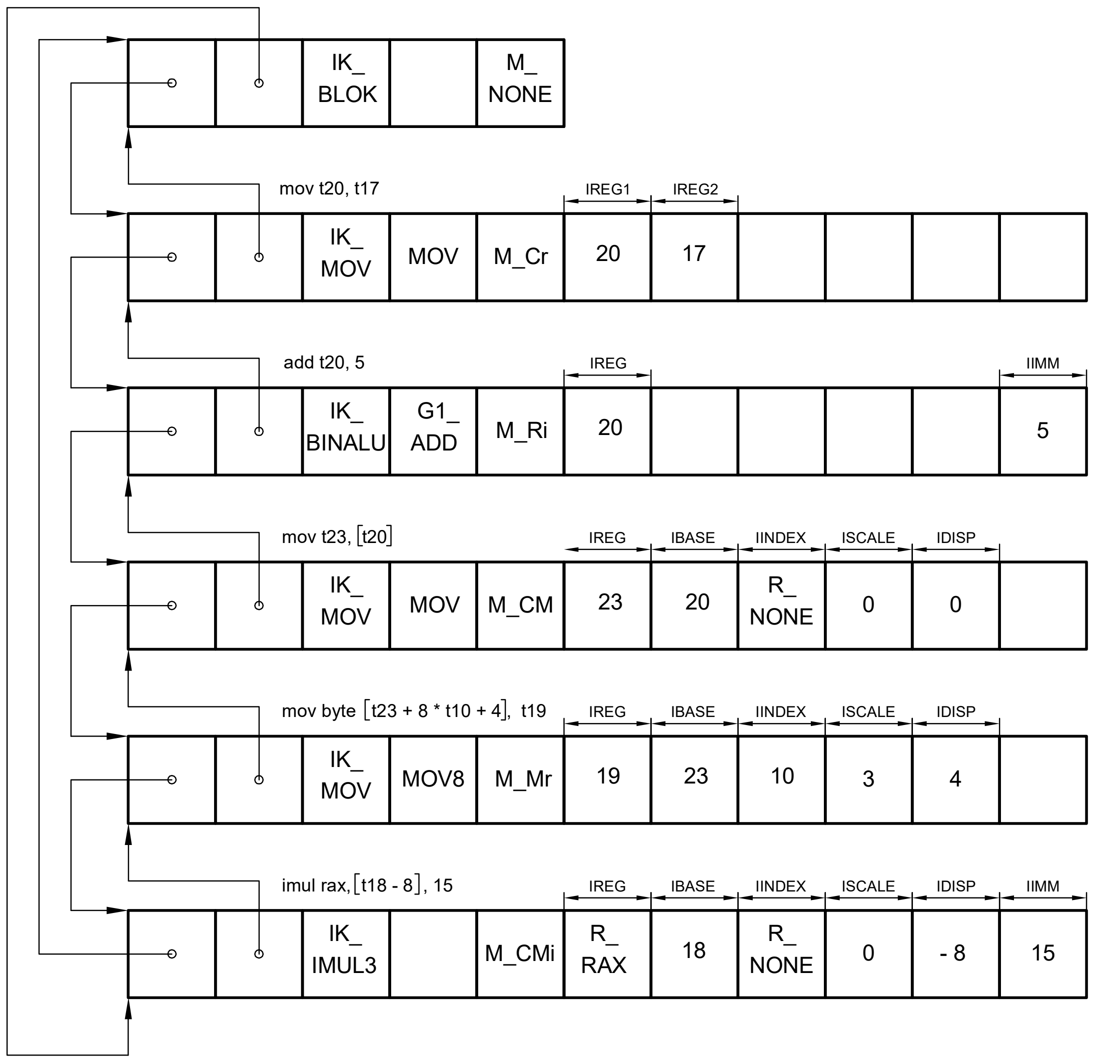

Figure 4.2. A machine block with a few x86-64 instructions linked in a circular linked list

The splitting can be realized in a single linear pass over all basic blocks in a function. For each block, if it has multiple predecessors, we check for each predecessor whether it has multiple successors—if it does, we have found critical edge. The edge is split by introducing a new basic block, which contains just one jump operation into the successor, and has one predecessor—the original predecessor of the edge.

Full implementation critical edge splitting in our implementation is shown below. The algorithm is rather simple and shows the use of our design and data structures. In particular, we do the critical edge splitting on the *middle end* representation. This is not only because splitting of critical edges is fully machine independent, but also because we currently don't allow all kinds of control flow changes in the back end representation.

A few key points in the implementation deserve a mention:

- Iteration over blocks is based on reverse postorder, by iterating over the precomputed postorder in reverse. New blocks are *not* added to the post order on the fly. We don't need to visit them in our algorithm, since they are created without critical edges.
- Since the postorder isn't updated during the run of the algorithm, it is updated after the algorithm finishes.

- Adding an operation to the end of a basic block can be realized with a `prepend_value` function, which prepends a value to a doubly linked list of values. Since the block is the head of the circular doubly linked list of instructions, anything prepended to it will become the last instruction.
- We have to iterate linearly to find a predecessor/successor to replace in the list of them.

```
void split_critical_edges(Arena *arena, Function *function) {
   for (size_t b = function->block_cnt; b--;) {
      Block *succ = function->post_order[b];
      if (block_pred_cnt(succ) <= 1)
         continue;

      FOR_EACH_BLOCK_PRED(succ, pred_) {
         Block *pred = *pred_;
         if (block_succ_cnt(pred) <= 1)
            continue;

         Block *new_block = create_block(arena, function);
         block_add_pred(new_block, pred);
         Value *jump = create_unary(arena, VK_JUMP, &TYPE_VOID, &succ->base);
         jump->parent = &new_block->base;
         jump->index = function->value_cnt++;
         prepend_value(&new_block->base, jump);

         FOR_EACH_BLOCK_SUCC(pred, s)
            if (*s == succ)
               *s = new_block;

         FOR_EACH_BLOCK_PRED(succ, p)
            if (*p == pred)
               *p = new_block;
      }
   }

   compute_postorder(function);
}
```

4.5 SSA deconstruction

In the middle end we use value-based SSA (introduced in section 3.4.1) and also ϕ-functions. In the back end we work with machine instructions and virtual and physical registers. Thus we need to map values to virtual registers (to be mapped to physical registers later) and also need to replace uses of ϕ-functions with copy (`mov`) instructions.

Assignment of virtual registers is simple in our representation. Because we already assign an index to a `Value` for printing and array indexing purposes (see section 4.3.1), we can use the integer indices directly as virtual registers. Since the virtual registers are assigned directly from SSA form, the virtual registers will obey the single assignment property and thus also correspond to *live ranges* for which we want to allocate registers.

In our design we will do SSA deconstruction on the value-based representation with method 1 from [Sreedhar et al., 1999] (see section 3.5 for more details). Doing it on the middle end value-based representation turns out to be more straightforward and is also how Sreedhar's method nominally works.

Sreedhar's method 1 consists of adding a copy instruction to each predecessor of the block holding the ϕ-operation and also an extra copy after the ϕ itself. Next all virtual registers involved in the ϕ-instruction itself are given the same virtual register and the ϕ-instruction can be safely removed.

In our implementation we operate with *values*, not instructions themselves. So realizing copy instructions is not possible in the strict sense. Though we can introduce *identity operations*. These are operations with single operand, that just copy the operand. Since we will also be using the `index` field of values as the virtual registers, we can assign the same virtual registers to all the identity operations (the to-be copy instructions), and also replace the ϕ-operation itself with a copy instruction, which copies from the same virtual register by inserting a dummy value with the right index.

Consider for example the following function `f`, which returns 4 or 3 depending on the truthiness of the first integer argument:

```
f:
        v0: int = argument 0
block0:
        branch v0, block2, block5
block5: block0
        jump block4
block2: block0
        jump block4
block4: block2, block5
        v4: int = phi 4, 3
        ret v4
```

We deconstruct the phi by introducing copies to block 2 and block 5 and changing the phi to be a copy itself. All the copies need to be based on the same index (which will become a virtual register):

```
f:
        v0: int = argument 0
block0:
        branch v0, block2, block5
block5: block0
        v6: int = identity 3
        jump block4
block2: block0
        v6: int = identity 4
        jump block4
block4: block2, block5
        v4: int = identity v6
        ret v4
```

Our value based SSA form always stays in SSA, since values can't be assigned. But by making the indices semantically meaningful, we can deconstruct the SSA with the right assignment of virtual registers and identity operations that translate to copy (`mov`) instructions.

Doing the SSA deconstruction on the SSA form has one significant advantage over doing it on the machine form—the control flow graph is fully built already, and we can easily just insert copies into predecessors. Doing it in the code generator would not be as straightforward. We plan our code generator to be very simple and single pass. Requiring copies in predecessor blocks becomes troublesome in such generator, because at the time of lowering a block with ϕ-operations the predecessor block may have not been translated yet and there may be no place for the copy insertion! Alternatively, we could flip the idea of SSA deconstruction, and instead of inserting copies into predecessors while processing the block, we can insert the copies to the predecessors, while lowering *them*—this is easier, since we base this on the existence of ϕ-operations in *successor* blocks in the middle end representation and don't need the successors to be already translated. However, in Sreedhar's method we have to introduce a new virtual register for the copies, and suddenly coordinating when and how the virtual register is allocated becomes more messy than doing it in the middle end IR.

We considered even doing the two step copying method (see section 3.5), which works well even when done when processing the predecessors—the two rounds of copies are fully self contained in the predecessors, no indexes have to be changed or coordinated elsewhere. However, as the two copy method puts a lot of unnecessary burden on register coalescing, we decided to go with Sreedhar's method 1 instead. Other methods are able to save even more on future coalescing by inserting the copies in a smarter way, by essentially doing the coalescing in the SSA deconstruction stage, but we haven't yet found the need to justify a much more complicated algorithm for not as that many benefits. If we ever do, [Boissinot et al., 2009]'s approach is the way to go.

4.6 Live range splitting

Currently we don't do any extra live range splitting. Deconstruction of ϕ-operations naturally introduces splits into a lot of places where they would be interesting—mainly loops.

Still, we found that sometimes a spill of a variable also spills distant, uses of the register. In these situations splitting could help to minimize the spilling damage. Splitting around loops and function calls seems like an addition that could in some cases improve the generated code. Especially for live-through registers, which are not split naturally by ϕ-operations.

4.7 Lowering

Code generation or *lowering* is the stage where we go from middle end (machine independent) representation of values to machine dependent instructions. We designed our code generator to be simple, with no case analysis. It receives as input a RISC-like value based IR and will transform it to RISC-like x86-64 machine instructions.

We mostly solved SSA deconstruction in section 4.5 and thus have already assigned virtual registers. The lowering step should then simply go over the values in the control flow graph one by one and translate them to corresponding instructions. For following analysis in register allocation, we will preserve the basic blocks, and as we don't have to add any new basic blocks, the translation is done one-to-one in terms of basic blocks. Not as much with instructions, where one middle end operation may require multiple x86-64 instructions.

In section about implementation of SSA deconstruction (4.5) we claimed that for values we simply use their index as the virtual register. So in order to transform an operation like the following addition:

```
v5: int = add v3, v4
```

We want to emit an `add` instruction adding virtual register 3 and virtual register 4 into virtual register 5. However, there is no such instruction on the x86-64. We only have available two address code instructions (see section 2.2.1 for more details), where the result is put into the first source register. We can only `add` virtual register 3 and virtual register 4 into virtual register 3[1]:

```
add t3, t4
```

To move the result into `t5` we could add a copy instruction after:

```
add t3, t4
mov t5, t3
```

However, to also preserve the original value of `t3` (since in general it may be needed elsewhere), we do the copy *before*:

```
mov t5, t3
add t5, t4
```

We do similar translations for most of the ordinary binary operations. Unary operations are very similar, we go from:

```
v3: int = neg v2
```

to:

```
mov t3, t2
neg t3
```

Instructions which have special register constraints are realized through copies to or from the required registers (see section 3.8.3.6), thus effectively we perform *live range splitting* (explained more thoroughly in section 3.8.3.5). For example shifts require the shift amount to be in register `cl`, i.e. the following:

```
v3 = shl v1, v2
```

needs to translate to:

```
mov t3, t1
mov rcx, t2
shl t3, cl
```

Another example is (signed) division which essentially requires the dividend to be 128 bits: lower 64 bits in `rax`, upper in `rdx`. Since we only need 64-bit division, we can move the 64-bit dividend into `rax` and sign extend it into `rdx` with the special purpose instruction `cqo` ("convert quadword to octoword"), which does exactly that. After the division, the quotient is in `rax` and we need to move it to a temporary as well:

```
v3 = sdiv v1, v2
```

to:

```
mov rax, v1
cqo
idiv v2
mov t3, rax
```

In a RISC-like fashion, we base our operations on registers and don't investigate whether we can use a more suitable addressing mode. One problem is, that in the middle end IR, we allow integer constants to be used directly in operations, i.e. this:

[1] Notice that we use the letter `t` as in "temporary" for virtual registers to distinguish them from values which are prefixed with letter `v`, however for readability and ease of implementation we keep the indices themselves mostly the same.

```
v7: int = add v6, 1
```

Here `v7` is an `Operation` whose first argument is another operation with index 6, and the second argument is a `Constant` (see section 4.3.1). Operations are embedded in concrete places in control flow graph and represent the *value* resulting from an operation, which is only valid in the function (and to be concrete speaking only in operations *strictly dominated* by the operation). But constants are not anchored anywhere in the control flow graph (at least in our implementation). There are two practical problems with assigning them virtual registers:

- Our virtual register assignments are not global across all functions functions, but only per-function.
- Even if we did for example assign the constant 4 the virtual register 4 across all functions, later we would have to assign it a *physical register*. Blocking a physical for storing constants is unacceptable, if only because much more constants can be used than there physical registers available.

As opposed to operations, constants are *always* available. Because they are also *compile time constants*, instead of assigning virtual registers to constants, we can materialize the constants to registers *on demand*, when needed—and each time into a new, *fresh*, virtual register. This way, we not only keep the constants close to their uses (allowing easier peephole optimization), but also keep the fresh virtual registers very short lived. This way we alleviate much of the need for *rematerialization of constants* in register allocation (see for example [Chaitin et al., 1981])—due to being short lived, they are not great candidates for spilling, and many constants are easily folded into other instructions through the use of better addressing modes.

For these reasons we don't put constants like integers into control flow graph like other operations, i.e. addition for two numbers wouldn't be represented as:

```
v1: int = constant 1
v2: int = constant 2
v3: int = add v1, v2
```

but as:

```
v3: int = add 1, 2
```

For many of the constants (like functions, globals and string literals) we can also choose either form, e.g. for functions:

```
v4: *(int) -> int = function f
[...]
v6: int = call v4, v5
```

But also:

```
v6: int = call f, v5
```

Materialization of constants has to be done every time we encounter the constant being used as an operand. As our code generator doesn't do any case analysis, it suffices to materialize the constant into a register. For example, we would translate

```
v3 = add 1, 2
v4 = call f, v3, 1, 2
```

to the following:

```
mov t10, 1
mov t11, 2
mov t3, t10
add t3, t11

lea t12, [f]
mov t13, 1
mov t14, 2

mov rdi, t3
mov rsi, t13
mov rdx, t14

call t12
```

Above, the first group of instructions realizes the addition including the materialization of constants. In the second group, the constants for the call are materialized—the address of the function is loaded into a register, just as well as the constants. In the third group, there are copies to registers imposed by the calling registers—i.e. first argument to `rdi`, etc. And finally in the last instruction the function is called indirectly through a function pointer.

The code generated by our simple code generator is very inefficient, but produces correct, obvious code, that can be improved upon in the peephole optimization stage. The peepholer is not only more suited to do case analysis, but it can also do it across the code expansions of more operations. For example, we could make the code generator generate direct function calls ("label calls") instead of indirect calls ("register calls"), but we want to transform indirect calls to direct even in the case they didn't seem as direct at first, but only appeared so after other (peephole) optimizations.

Even use of immediates in instructions isn't without case analysis: generally only 32-bit immediates (which are sign extended) are allowed for instructions. For example the following the additions have different possible *optimized* versions:

```
v7: int = add v6, 2147483647

v9: int = add v8, 2147483648
```

```
lea t7, [t6 + 2147483647]

mov t10, 2147483648
mov t9, t8
add t9, t10
```

The code generation is mainly driven by a function called `translate_value`, which is called in turn for each value in a basic block. The function is responsible for translating the values anchored in the control flow graph, i.e. mostly the *operations*.

```
void translate_value(TranslationState *ts, Value *v) {
   Oper ops[256];
   Value **operands = value_operands(v);
   size_t operand_cnt = value_operand_cnt(v);
   for (size_t i = 0; i < operand_cnt; i++)
      ops[i] = translate_operand(ts, operands[i]);

   Oper res = v->index;
   switch (v->kind) {
   case VK_IDENTITY: add_copy(ts, res, ops[0]); break;
   case VK_ADD:      translate_binop(ts, G1_ADD, res, ops[0], ops[1]); break;
   case VK_SUB:      translate_binop(ts, G1_SUB, res, ops[0], ops[1]); break;
   case VK_SHL:      translate_shift(ts, G2_SHL, res, ops[0], ops[1]); break;
   case VK_SAR:      translate_shift(ts, G2_SAR, res, ops[0], ops[1]); break;
   case VK_NEQ:      translate_cmpop(ts, CC_NE,  res, ops[0], ops[1]); break;
   case VK_SLT:      translate_cmpop(ts, CC_L,   res, ops[0], ops[1]); break;
   [...]
   case VK_JUMP: add_jmp(ts, ops[0]); break;
   [...]
   }
   }
}
```

Before any operation is translated, the *operands* are translated with the function `translate_operand`—mainly translation of operands involves materialization of constants. Then for each operation we translate it into a sequence of instructions using the operands in registers and writing the result into a register. Our naming convention names simple instruction constructors as `add_`⟨*instruction*⟩ and helper functions that add multiple instructions `translate_`⟨*kind*⟩.

For example, the copy instruction has the following constructor, which initializes all operands as well as the kind, subkind and mode:

```
void add_copy(TranslationState *ts, Oper dest, Oper src) {
   Inst *inst = add_inst(ts, IK_MOV, MOV, M_Cr);
   IREG1(inst) = dest;
   IREG2(inst) = src;
}
```

And translation of binary operations is as follows:

```
static void translate_binop(TranslationState *ts,
      X86Group1 op, Oper res, Oper arg1, Oper arg2)
{
   add_copy(ts, res, arg1);
   add_binop(ts, op, res, arg2);
}
```

There are not that many different kinds of operations from the perspective of the x86-64, so a few helpers suffice to cover most operations.

4.7.1 Operands

The translation of operands with the `translate_operand` function is reproduced below in full for clarity:

```
Oper translate_operand(TranslationState *ts, Value *operand) {
   Oper res;
   switch (operand->kind) {
   case VK_BLOCK:
      res = operand->index;
      break;
   case VK_FUNCTION:
   case VK_GLOBAL:
   case VK_STRING: {
      size_t label_index = add_label(ts->labels, operand);
      res = ts->index++;
```

```
            add_lea_label(ts, res, label_index);
            break;
        }
        case VK_CONSTANT: {
            Constant *k = (void*) operand;
            res = ts->index++;
            add_mov_imm(ts, res, k->k);
            break;
        }
        case VK_ALLOCA: {
            Alloca *alloca = (Alloca *) operand;
            res = ts->index++;
            add_lea(ts, res, R_RBP, alloca->stack_offset);
            break;
        }
        default:
            res = operand->index;
            break;
    }
    return res;
}
```

Most kinds of values (i.e. all operations) don't need anything special—our invariant already puts them into registers. But constants like integer constants or addresses of functions, globals or strings need to be materialized. As they are not assigned one static register, we allocate a fresh virtual register with a simple increment of a counter. The following are the instructions that get introduced with translation of operands:

```
lea t2, [f] ; function
lea t2, [G] ; global
lea t2, [$str3] ; string
mov t2, 34 ; constant
lea t2, [rbp-32] ; alloca
```

Basic blocks are represented with values in our implementation, but they are not *first class* values. We don't allow the address of a basic block to be taken.[1]

4.7.2 Stack slots

Currently stack slots are handled with `Alloca`, which is a weird mix between an actual instruction (akin to C's `alloca`) and a constant. We embed `alloca`s in control flow, and not hold them separately, like for example function arguments. But we also evaluate them as constants, i.e. they don't actually allocate memory dynamically.

We might fully switch to either the dynamic operation or static constant in the future. For now it works for ordinary TinyC programs.

4.7.3 Prologue and epilogue

For prologue, we choose to always establish a base pointer `rbp`. The one free register that we could gain by not using it as a base pointer is not worth it, in our opinion. Establishing the stack frame is very useful for debugging and to human readers of the code.

[1] Some implementations of the C programming language, like GCC, allow taking the address of a label: `https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html`. With sufficient support in the front end, our implementation could easily support this by translating basic blocks similarly as functions. After all, in assembly they are just labels, and the only difference is in calling conventions.

Hence all our functions are wrapped in:

```
push rbp
mov rbp, rsp
sub rsp, 42
[...]
mov rsp, rbp
pop rbp
ret
```

Additionally, in the prologue, we reserve the needed stack space with a sub instruction. As we don't yet known how much stack space will be required (because spills may be introduced to the stack), we just add a dummy constant to be replaced later. We can't use zero, because peephole optimizer would optimize the instruction as unuseful.

Saves and restores of all callee saved registers were already explained elsewhere. But importantly we also copy the arguments from the calling convention registers to the virtual registers associated with the `Argument`. E.g. for the first argument with a value index 17, we might emit the following copy instruction:

```
mov t17, rdi
```

This is yet another form of live range splitting. Though the situation is a little bit different with arguments passed on the stack. There we issue a load instruction instead. E.g. for the 7th argument:

```
mov t23, [rbp+16]
```

4.7.4 Narrow types

TinyC supports `char`s as well as `int`s. In our implementation we chose to go with C's semantics, where most narrow integer types get promoted to `int` before any other operation.

As a result, in our type system and implementation, there are only two instructions which know about 8-bit `char`s—loads and stores. For this we introduce special subkind for 8-bit store (`MOV8`) and a sign extending load (`MOVSX8`). The stores naturally restrict the range through the use of 8-bit register parts (e.g. `al`). Sign extension is able to read a byte from memory and store it into a full 64-bit register.

4.7.5 Signedness

Our back end fully supports both signed and unsigned operations. This is currently mostly unused, since TinyC the language has only signed types.

Interestingly, adding support for unsigned operations in the back end is just a matter of adding the 4 unsigned comparisons to `translate_value`, as well as unsigned division:

```
case VK_UDIV:
   translate_div(ts, res, ops[0], ops[1], SK_UNSIGNED, DR_QUOTIENT);
   break;
case VK_UREM:
   translate_div(ts, res, ops[0], ops[1], SK_UNSIGNED, DR_REMAINDER);
   break;
case VK_ULT:  translate_cmpop(ts, CC_B, res, ops[0], ops[1]); break;
case VK_ULEQ: translate_cmpop(ts, CC_BE, res, ops[0], ops[1]); break;
case VK_UGT:  translate_cmpop(ts, CC_A, res, ops[0], ops[1]); break;
case VK_UGEQ: translate_cmpop(ts, CC_AE, res, ops[0], ops[1]); break;
```

Instead of performing sign extension with `cqo`, unsigned division can just zero out the upper 64-bits of the dividend (`rdx`).

4.8 Peephole optimization

In our compiler, peephole optimization serves the roles of both the instruction selection and the classic peephole optimization that runs as the last step of the backend for code cleanup. This is because we chose to use a single back end representation—machine instructions. Our peephole optimizer shall look at sequences of instructions and replace them with *better* alternatives, where better is often faster, shorter, or even just more canonical to allow subsequent optimizations. We should also be careful about optimizations that transform code that *doesn't* allow subsequent optimizations.

In the next few subsections, we will look at optimizations applicable to x86-64 machine instructions (or in general) and also into the implementation in our back end (section 4.8.5).

4.8.1 Local optimizations

Local optimizations are the classic peephole optimizations based on small windows *peepholes* into instructions. They are very limited, since they have only very local knowledge about the code.

As mentioned when discussing coalescing (section 3.8.3.4), the register allocator tries to assign virtual registers involved in the same move instruction the same physical register. In that case we can end up with instruction like:

```
mov rax, rax
```

On the x86-64, copy instruction where the source is the same as destination can be deleted freely, because they have no effect—they don't even set flags.

For an implementation of this particular peephole pattern, we need to match the *exact opcode*. In this case we need to match a `mov` from register to register. In our representation of instruction (described thoroughly in section 4.3.2), we split the opcode information into three parts: *kind*, *subkind* and *mode*. This split is especially targeted at peephole patterns, where we can get a lot of patterns sharing the same kind, but with different subkinds, or the patterns can be applicable to different instructions with the same mode.

Kind ("instruction kind", `IK`) for this pattern would be `IK_MOV`, subkind `MOV`[1] and mode `M_Cr` (i.e. first register only written, second register only read). And we also need to check whether the two registers are the same. These checks don't need to occur in the order described here—as long as they all succeed, they can be done in any order, however filtering first based on the kind and subkind can quickly filter out instructions not worth checking.

The entire implementation of the pattern could be the following:

```
if (IK(inst) == IK_MOV && IS(inst) == MOV && IM(inst) == M_Cr
      && IREG(inst) == IREG2(inst)) {
   inst->prev->next = inst->next;
   inst->next->prev = inst->prev;
}
```

Here, the macros `IK`, `IS` and `IM` respectively allow terse access to the `kind`, `subkind` and `mode` fields of the instructions (see section 4.3.2.2 for more details). If the pattern matches, we can remove the instruction by unlinking it from the circular doubly linked

[1] There are multiple variations of the `mov` opcode, in particular for different sizes (i.e. 8-bit vs 64-bit loads/stores).

list by pointing the previous node's next field to the next instruction and likewise for the `prev` field of the next instruction.

There are more single instruction peephole patterns that are possible. For example a comparison of a register with a 0:

```
cmp t12, 0
```

is the same as using `test` instruction (which performs bitwise and of the two operands and sets flags based on the result) on the register with itself:

```
test t12, t12
```

This saves 4 bytes on the immediate 0. The implementation of the pattern can look like this:

```
if (IK(inst) == IK_BINALU && IS(inst) == G1_CMP && IM(inst) == M_ri
      && IIMM(inst) == 0) {
   IS(inst) = G1_TEST;
   IM(inst) = M_rr;
   IREG2(inst) = IREG(inst);
}
```

We check for kind, subkind, mode and then the special properties we are looking for, in this case that the 32-bit immediate is zero. To realize the replacement of the instruction we don't have to allocate a new one and free the old one. We can use the existing instruction and modify it in place. This saves as quite a bit of relinking we would have to do, but more importantly it allows us to keep some fields *unchanged*, which is really handy for some patterns. Here we for example don't need to change the first register—it stays the same for both instructions. In more complicated patterns this can be more interesting—often we e.g. don't care about how a memory operand looks, we can just pass it through.

When considering peephole patterns we don't need to just find just *useful* patterns, they need to be *reachable* as well. For example, the transformation of comparison with zero to `test` presented above is useful (produces shorter instruction encoding), but it as of now, it is not reachable—our code generator never generates comparisons with immediates.

In this case, immediate operands are in fact very common, because they can be created with simple peephole optimization like this one:

```
mov t13, 0
cmp t12, t13

cmp t12, 0
```

This pattern is more general—certainly we don't need it to be limited to just zero. But we need to consider the fact, that on the x86-64 immediate operands can be at most 32-bit signed values, so "small" positive and negative immediates (which are the most common ones) are applicable. The pattern is also not limited to the `cmp` instruction—it will also work with `add` or `sub`. All of these are conveniently in the `IK_BINALU` kind, since they have very similar addressing modes. But the mode is different—`cmp` like `test`, but unlike e.g. `add` *doesn't* write to any register—it just changes the flags. Thus we don't need to match for any particular *subkind*, but we need to check for one of the two *modes*. Here an implementation could look like this:

```
if (IK(inst) == IK_BINALU && (IM(inst) == M_Rr || IM(inst) == M_rr)
      && IK(prev) == IK_MOV && IS(prev) == MOV
```

```
        && IM(prev) == M_CI && IREG(prev) == IREG2(inst)
        && pack_into_oper(get_imm64(prev), &IIMM(inst))) {
    inst->mode = IM(inst) == M_Rr ? M_Ri : M_ri;
    IREG2(inst) = R_NONE;
}
```

As hinted above, we check the current instruction (`inst`) for the right kind and mode, but we also investigate the previous instruction (here designated by the local variable `prev`). We expect it to be a `mov` with mode `M_CI`. The big letter `I` signifies a 64-bit immediate—moves of immediates into registers are one of the very few instructions where 64-bit immediates are allowed. Hence before applying this pattern we also need to check whether the immediate from the `mov` instruction actually fits into 32-bit with the sign bit. Here we do that, with helper functions which extract the 64-bit immediate from two 32-bit `Oper` slots, and then try to fit it into the 32-bit `Oper` of the arithmetic instruction. The function returns a boolean indicating success, but also performs a side effect, for this reason we call it as last check. If successful, we update the mode of the instruction accordingly—we will use mode with `i` for the second operand (since it is now a 32-bit to-be sign extended immediate), and we preserve the use and def property on the first register. Also, since generally we keep unset fields set to zero, we also set the (now unused) second register field to 0 (or the symbolic constant `R_NONE` which is defined to be 0)—we don't need to do this, because as long as the mode doesn't need the slot, it is not going to be read, but the zero-initialized property is sometimes useful and we chose to preserve it.

Unfortunately, the pattern shown above is still not very useful—it expects two the immediate move and arithmetic instructions to immediately follow each other. But for example subtractions are lowered into code like this:

```
mov t13, 10
mov t14, t12
sub t14, t13
```

then the copy in the middle prevents the pattern to match. Though if `t14` and `t12` get coalesced:

```
mov rax, 10
mov rcx, rcx
sub rcx, rax
```

the pattern will match after the redundant copy is optimized out:

```
sub rcx, 10
```

Though the code is better, and uses a better addressing mode, we missed an optimization—by waiting until after register allocation, we had to allocate a physical register for the short lived temporary. This could have caused a long lived temporary to get spilled or at least makes the register allocation problem harder by keeping more interferences. Thus we really do want to make optimizations in the first peephole pass *before* register allocation. Here we can notice the pattern with the move in the middle and introduce a three instruction pattern. The new pattern should probably check fully that the middle instruction is a `mov` in the form we expect, or at least it has to make sure, that the register with the immediate isn't *overwritten* in the middle instruction, which would make the optimization invalid.

Apart from register or immediate operands, instructions can also have one memory operand. Instructions often use operands in memory—C variables are nominally stored on the stack, just like spilled registers. Addresses of variables on the stack (through `alloca` instructions) are *constants* in our implementation, hence are often very close to their use, making it possible for simple peephole patterns to optimize the memory access. For example, a load from a local variable might look like this:

```
lea t25, [rbp-24]
mov t26, [t25]
```

and can be easily optimized to this:

```
mov t26, [rbp-24]
```

This optimization is nicely possible, because the address computation in the `lea` ("load effective address") instruction uses the same memory addressing as all other instructions referencing memory. Similar memory addressing optimizations are also applicable for stores and operands of arithmetic instructions.

Peephole optimizations don't only involve simple identities or addressing mode changes. Opportunities arise for example for elimination of a load from just stored address:

```
mov [G], t27
mov t28, [G]
```

Instead of loading from the global variable `G`, it would seemingly be possible to use the register `t27`. However, all uses of `t28` currently refer to `t28` and we can't easily changed them without additional bookkeeping. Merging the registers into one practically is just like coalescing, and we can encourage the register allocator to coalesce just by rewriting the load into a copy:

```
mov [G], t27
mov t28, t27
```

The advantage to leaving the coalescing to the register allocator is, that it has the knowledge about interferences and will not combine `t28` with `t27` if they *interfere* (see section 3.8.4.3). Virtual registers which interfere are *live* at the same time at some point, and can't share a single physical register. Our limited view in the peephole optimizer doesn't allow us to tell if the optimization would be safe, so we always stay with safer copies if applicable.

4.8.2 Flag based optimizations

Well known arithmetic identities like addition of zero or multiplication by one are often folded in the middle end. Though, with an optimizing back end, opportunities for such optimizations often arise again. Hence our peephole optimizer should be able to do them as well. They are not as straightforward though. Take as an example addition with zero:

```
add rax, 0
```

While the register `rax` isn't changed by the instruction at all, the *flags* register gets updated based on the *result* of addition—in this it will reflect `rax`. If a later instruction depends on the flags, we can't just delete this instruction.

Similar need for flags prevents the use of `lea` for arithmetic, because unlike arithmetic instructions the `lea` instruction *doesn't* set flags. I.e. the following addition:

```
mov t26, t18
add t26, t34
```

can only be optimized to the below `lea` if the flags are not observed after the `add`:

```
lea t26, [t18+t34]
```

Normally our code generator doesn't generate `lea` instructions except for materialization of constants (see section 4.7). However, as shown in the previous section (section 4.8.1), we are able to transform uses of `lea`s in loads and stores or other instructions that allow memory operands. So transforming address calculations involving `add` or `imul` into `lea` instructions is important to allow these better addressing modes. Transforming even ordinary arithmetic (i.e. not address calculations) into `lea` instructions is also almost always an improvement—`lea` unlike most other instructions can write a different register than any of the read registers, so its use may constrain register allocation less.

As our code generator doesn't do any case analysis, it inserts explicit `cmp` instructions to set the flags. Thus we could perform the optimizations described above freely. But then we would have to give up the optimizations that are able to *use* the flags. For example a loop might be decrementing a register until it becomes zero:

```
sub rax, 1
test rax, rax
jz .L3
```

Using `test` to set flags according to `rax`'s value is redundant, since the flags are already set by the previous arithmetic instruction. The instruction sequence above is not unreal—we have shown how optimizations can replace comparison with zero to `test` with itself in section 4.8.1.

Ideally we would like to support both kinds of optimizations—removing redundant settings of flags when the flags are already set by arithmetic instructions, as well as making use of instructions not setting the flags if the flags are not needed. For this we need to track whether flags set by an instruction are *observed*. We do that by introducing three flags to each instruction:

```
struct Inst {
   [...]
   bool writes_flags;
   bool reads_flags;
   bool flags_observed;
   [...]
};
```

And then compute the `flags_observed` property based on how the instructions read and write flags in a single backwards pass over a basic block:

```
bool flags_needed = false;
for (Inst *inst = block->insts.prev; inst != &block->insts; inst = inst->prev) {
   inst->flags_observed = flags_needed;
   if (inst->writes_flags)
      flags_needed = false;
   if (inst->reads_flags)
      flags_needed = true;
}
```

This way instructions which set flags needed in the future, as well as instructions through which needed flags pass through are marked. More complicated data-flow analysis is not needed, since we don't ever assume anything about the state of flags at basic block starts and can thus assume the flags are not needed at the end of a basic block.

The boolean flags on `Inst` are flexible, but perhaps unnecessarily since whether the instruction writes or reads flags could be derived from the instruction *kind*. But since in our implementation instructions had spare space for the two flags due to alignment, we didn't yet find the need for such improvement.

Now with sufficient information, we may realize simple arithmetic identities like addition of zero or multiplication by one: depending on whether the flags are observed, we can either remove the instruction completely, or change it to `test` of the register with itself. `test` is a better way of setting the flags then an arithmetic instruction, if only because it doesn't involve an immediate.

The information about flag observation can be used for even more optimizations. For example, `cmp` and `test` instructions can be removed if the flags set by them are not observed:

```
cmp t3, t5 // can be deleted if flags not observed
```

Some instructions are problematic with regards to flags. For example `inc` and `dec` instruction don't set the carry flag, but do set other flags and shift instructions (like `shl`) don't set the flags when the shift amount is equal to zero. The special behavior of `inc` and `dec` isn't that problematic, since we currently don't use the carry flag for anything. Shifts are more problematic, since if we expect them to set flags and optimize based on that like above, we might actually get into situation where the flags are not set and instead a previous incorrect value of the flag register would be read. For this reason we don't set `writes_flags` for shift instructions. This way, if flags are needed after the instruction, the peephole optimizer will be forced to leave the `cmp` instruction inserted by the code generator. This will then be marked as pass-through of flags by our algorithm above, though this isn't a problem with code patterns generated by our generator, where we always set flags and read flags in the same expansion, not across expansions.

4.8.3 Use-def based optimizations

In section 4.8.1 we showed an addressing mode optimization, which took an immediate move instruction or `lea` loading an address of local or global variable and folded the constant into other instruction. For example from:

```
mov t13, 0
cmp t12, t13
```

to:

```
cmp t12, 0
```

However, there are a few problems with this optimization:

- The optimization is actually incorrect. We can't remove the definition of `t13`, since there may be other uses of the register. In practice, since during code generation we *materialize constants* for each use (see section 4.7), there are *no* other uses of the constants. But other optimizations can change that.
- It is not powerful enough. The optimization only takes place when there are two (physically) consecutive instructions.

We can improve on both points by tracking *uses* and *definitions*. We call this *use-def* based peephole optimization. Our approach and idea are pretty simple: for each register we track the number of definitions, the number of uses and if there is only one definition, then we also track the defining instruction. These properties are precalculated once, before the peephole pass, and in the implementation we actually do it as part of the backwards pass over blocks where we track flag information.

This information about the number of definitions and uses allows us to remove instructions that define a register with no uses (provided that the instruction doesn't have other side effects). Such peephole pattern may be even as abstract as this:

```
if ((IM(inst) == M_CI || IM(inst) == M_Cr || IM(inst) == M_CM
      || IM(inst) == M_Cn) && use_cnt[IREG(inst)] == 0) {
   def_cnt[IREG(inst)]--;
   for_each_use(inst, decrement_count, use_cnt);
   inst->prev->next = inst->next;
   inst->next->prev = inst->prev;
}
```

Here we check the *mode* of the instruction and if is one of those that define the first register, and the register has no uses, then we can remove the definition by unlinking the instructions and decrementing the definition count of the register. To allow further optimizations, we also decrement the use counts of the registers used in this instruction—which may in effect lead to them being unused—though the implications of this on the peephole optimization are only discussed later, in section 4.8.5.

The tracking of the single definition is more interesting for thorough addressing mode improvements. For example, we can for an instruction in one addressing mode (say using only registers) check, whether e.g. the second register is defined as an immediate that fits into 32-bits, or a memory location computed using `lea`, etc. and use these definitions directly in the instruction with a better addressing mode. Additionally, if the original register with the constant is used only once, the definition can be deleted. With a right set of functions that abstract the check for constant definition and the folding, we can specify the peephole optimization patterns in a pretty terse way. Just for illustration:

```
if (IK(inst) == IK_MOV && IS(inst) == MOV && IM(inst) == M_Mr
      && try_replace_by_immediate(mfunction, inst, IREG2(inst))) {
   IM(inst) = M_Mi;
}
```

Above, if the instruction is a store instruction (moving a value from register to memory location) and the register is actually a 32-bit constant, then we can change the addressing mode to “memory-immediate”. The helper function takes care of checking the (single) constant definition and packing the immediate into `IIMM(inst)`, which can of course fail if the constant cannot be expressed as a 32-bit signed number. And just for completeness, the folding of memory locations can be done like this:

```
if (IK(inst) == IK_MOV && IS(inst) == MOV && IM(inst) == M_CM
      && try_combine_memory(mfunction, inst)) {
   // all work done by `try_combine_memory`
}
```

The above pattern can fold the memory location of the address in a load instruction. I.e. to go from the following:

```
lea t25, [rbp-24]
mov t26, [t25]
```

to:

```
mov t26, [rbp-24]
```

The function `try_combine_memory` is actually pretty involved, because it not only allows folds of constants, it tries to combine any two memory locations together. The function doesn't care about the actual addressing mode of the instruction the peephole executes on (i.e. the fact that it is a load in the example above), it only looks at the memory specification. If the *base* register used in the specification (`t25` above) has only one unique definition, which is a `lea` instruction and the *base* register of the `lea` (`rbp` above) is either `R_NONE` (meaning that RIP-relative, i.e. label addressing is used) or `rbp`[1] or other register with *one unique definition* then it will try to combine the base, index, scale and displacement fields of the two memory locations. So for example, even these two memory locations can be folded:

```
lea t27, [a-4]
mov t28, [t27+8]
```

and these two as well:

```
lea t29, [4*t25] // say t25 has only one definition
mov [t29+16], 1
```

The implementation currently doesn't support all possibilities that *could* be optimized. But sufficient number of them found in real programs using the advanced addressing modes are supported.

Similar helper function is used to transform `lea` instructions loading function addresses and indirect calls into direct calls:

```
if (IK(inst) == IK_CALL && IM(inst) == M_rCALL
      && try_combine_label(mfunction, inst)) {
   IM(inst) = M_LCALL;
}
```

i.e. this goes from:

```
lea rax, [function]
call rax
```

to:

```
call function
```

4.8.4 Inter-block optimizations

During code generation (section 4.7) we lowered jump and conditional jump operations at the end of basic blocks into `jmp` and `jcc` instructions. Depending on how the blocks are ordered, some of the instructions are unnecessary—unlike our middle end representation, which needs explicit jumps from each basic block, conditional jumps in machine code fall-through to the next instruction if the condition is not satisfied. This leaves opportunities for peephole optimization.

It is no longer enough to consider instructions alone—we need to also consider which block is the following one. Thus the optimizations depend on particular linearized *order* of the blocks. We will come back to the issue of block ordering in section 4.8.5, but for

[1] We have to special case `rbp`, because it actually has two definitions—one saving it in the prologue and one restoring it in the epilogue.

now it suffices to assume that when performing peephole optimizations at the end of a block, we know which block is the following "fall-through" block.

The most straightforward optimization possible, is to remove an unconditional jump to the following block:

```
   [...]
   jmp .BB5

.BB5:
   [...]
```

The `jmp` instruction above is always redundant. This also applies to unconditional jumps which result from translation of *conditional jumps*:

```
   [...]
   jge .BB4
   jmp .BB3

.BB3:
```

If the following block is not the target of the unconditional jump (`jmp`), but of the conditional one (`jcc`), we can still perform the optimization, but we need to *invert the condition*:

```
   [...]
   jge .BB7
   jmp .BB8

.BB7:
```

```
   [...]
   jl .BB8

.BB7:
```

Here we changed "greater-or-equal" condition to "less-than". This is done simply by changing the subkind of the instruction, since the subkind is the x86-64 condition code. There are 16 condition codes and inverse of a condition code can be found simply by inverting the least significant bit of the condition code.

It would seem that when the jump instruction is optimized out and implicit fall-through is used, then the *basic block label* can also be deleted. But this is only possible when there are *no other* jumps to the label—unconditional or not. Deleting the label would essentially mean a merge of two basic blocks, and basic blocks can only be entered at the start, so the merge of basic blocks is only possible if there isn't a jump into the "middle".

Even when the merge is possible, we can no longer speak about basic blocks, because there could be jump instructions in the middle—notably, if a block ended with a conditional jump compiled into `jcc` and `jmp` and `jmp` gets optimized away and the block gets merged with the following fallthrough block, the `jcc` instruction remains as an exit point in the middle of the block. The block is no longer basic. For example if we optimize from:

```
.BB6
   jge .BB7
   jmp .BB8

```

```
.BB7:
   mov t17, t15
   jmp .BB8

.BB8:
   [...]
```

To:

```
.BB6
   jl .BB8
   mov t17, t15
.BB8:
   [...]
```

We see how the label `.BB7` corresponding to block 7 gets eliminated, because blocks 6 and 7 are merged. The resulting merged block 6 is no longer basic—it can be left in the middle. Some distinction of *blocks* is still useful, since they still correspond to sequences of instructions which are executed after each other, even if the blocks are not basic. It is not a problem from the perspective of peephole optimization, which optimizes physically neighbouring instructions. There we can take advantage of the fact that the sequence is not entered in the middle, and we don't mind that it can be left in the middle.

Additionally, following up to the example above, we can notice that the whole purpose of the conditional jump above is to skip the copy instruction. On the x86-64 architecture the same behavior can be achieved with a `cmovcc` instruction with an inverted condition code:

```
.BB6
   cmovge t17, t15
.BB8:
   [...]
```

This again, may leave the block 8 mergeable into block 6 if there are no other jumps to it. This example is very close to reality—similar short blocks conditionally entered, unconditionally left, and with zero or more copy instructions in them are created by critical edge splitting (see section 4.4) and SSA deconstruction (see section 4.5). Coalescing is often able to assign the virtual registers the same *physical* register and the copies are often removed. By removing coalesced copies, transforming simple jumps around copies into `cmovcc` and merging blocks through peephole optimizations we can often get straight line code that is even more amenable to peephole optimization.

For each block we track the number of references to it. When last reference of the block is removed from physically preceding block, the two blocks can be merged together.

4.8.5 Implementation

In the previous subsections we presented intra-block, but also inter-block peephole optimization patterns and how they can be matched and applied with our representation of instruction. In this section we solve the problem of putting the peephole optimization together, i.e. how to apply subsequent optimizations even on the same instruction, how to represent and move the peephole sliding window, etc.

Considering only intra-block optimization first, we want to go through the instruction sequence, try to match a pattern, apply the optimization and to readjust the peephole window according to what the optimization does to the instructions. Our patterns are also not of one uniform size: we have patterns that match on a single instruction, but also patterns matching up to four instructions.

What works fairly well, is to iterate over the instruction sequence in forward, while having a notion of the *current* instruction. When iterating over the instructions we root all the peephole patterns to the current instruction, such that the current instruction always constitutes the very end of the peephole window—matching on the context is done only on on *previous* instructions. If a pattern matches, it executes and modifies the instructions and perhaps even reorders them. Then, a new instruction is set as the current one and new matching is tried again with the same patterns. Since the *current* instruction is the last one in the peephole window, each optimization should set the current instruction to be the instruction most further back on which peephole optimizations are worthwhile, i.e. the instruction furthest back, which *changed*. This is because changed instructions can allow match of different patterns, so it is beneficial to go back and try them.

This construction has another advantage—it may even be used in an online way. For example, each time the code generator appends a new instruction, the peephole machine described above could be run on the last instruction, optimizing the tail of the produced code.

A state machine would be a great match for this kind of problem. The state machine could in its states remember the previous instructions encountered, and provide efficient matching and lookup of patterns to execute. In practice, the patterns, especially the use-def based (section 4.8.3) don't fit into this model that well. The fact that we can go back a few instructions would also mean a big number of states keeping the context. Additionally, the predicates used for matching the instructions are not always straightforward and often benefit of being coded in C, like shown in previous sections. Ultimately, creating a state machine generator also doesn't pay off that much until multiple target architectures are needed, so we settled on ad hoc pattern matching and optimization with C code.

In the implementation, we iterate the instruction sequence, try to match each pattern in turn, and, if any of them matches, its associated code executes and also sets the new current instruction, which we try matching next. If no pattern matches, we set current instruction to the instruction following the current one.

This is how it looks like in the actual implementation:

```
Inst *inst = mblock->insts.next;
while (inst != &mblock->insts) {
   if (⟨pattern 1⟩) {
      ⟨rule 1⟩
      inst = ⟨new current 1⟩;
      continue;
   }

   if (⟨pattern 2⟩) {
      ⟨rule 2⟩
      inst = ⟨new current 2⟩;
      continue;
   }

   [...]

next:
   inst = inst->next;
}
```

The pattern matching and rule execution is exactly like hinted in previous sections. If any pattern matches, new `inst` (current instruction) is set to go back sufficiently and the pattern matching "restarts". Otherwise the process continues with the next instruction.

The choice to do peephole optimization on instruction sequences limited to blocks instead of doing it on whole functions is not an obvious trade-off. In our architecture we do peephole optimization twice—before and after register allocation. For register allocation (due to liveness analysis, section 4.9.1) as well as flag analysis (section 4.8.2) we want to have *basic blocks*. But, for peephole optimization itself, we don't care about basic block that much. We can have either blocks that are not basic as hinted in section 4.8.4, or just a single instruction sequence for a whole function. Though still with a single instruction sequence we need a way to do jumps, which could be represented with *label* nodes, similar to what is done in [Wulf et al., 1975]. By having a linked list of instructions and labels, we could iterate over the whole function and not handle blocks as a special case—being represented as labels they would be part of normal patterns. After a label would become unreferenced, it would be deleted and instructions formerly separated would become adjacent and eligible for peephole optimization.

However, since we do a round of peephole optimization before register allocation's liveness analysis, we would need to derive information about basic blocks for it specially. Then we would have to be careful about keeping the basic block boundary representation valid even throughout the spill stage, which inserts new instruction (which depending on the boundary representation can be problematic e.g. if a load is inserted *before* the first instruction in a basic block) or we would have to recompute basic block information after spilling. Another problematic point for some representations of basic blocks is the fact, that liveness analysis needs to iterate over the blocks *backwards*, as well as keep them in a worklist for fast data-flow analysis (see section 4.9.1).

In our approach we chose to keep middle end basic blocks tied to machine blocks. Control flow graph from the middle end representation is used even in the back end, and merging of basic blocks is only done in the second round of peephole optimization (after register allocation), which is not followed by any analysis needing basic blocks. This works fairly well in that we don't introduce many different or even temporary intermediate representations, but still allows inter-block peephole optimization, though it has to be handled outside of the main loop shown above, which is executed for each basic block. The separate handling of inter-block optimizations looks roughly like this:

```
if (⟨there is a following block⟩) {
   if (⟨inter-block pattern 1⟩) {
      ⟨inter-block rule 1⟩
   }

   [...]

   if (⟨the following block is not referenced⟩) {
      ⟨merge following block into the current one⟩
      inst = ⟨first instruction of following block⟩;
      goto next;
   }
}
```

4.8.6 Practical findings

All in all, the peephole optimization seems to do fairly well. Early deletions of instructions are particularly important for getting good results out of peephole optimizations. In particular, dead instruction removal should be done as soon as possible, in the pass, not for example left to the next pass. These removals can bring closer instructions that would have otherwise not been covered by the same peephole window.

There are also a couple of disadvantages to our peephole optimization design:

- Patterns are written by hand which is tedious.
- Only patterns noticed by a human are implemented.
- The potential of use of data-flow is not fully realized yet, because deletions of constants far away from current instruction can still result in new cascading improvements.

Improvements on all points are of course possible, but require quite a different approach and are thus not yet implemented.

4.8.6.1 Copy propagation

There are still interesting patterns that have been found and implemented by hand. For example, the following works surprisingly well:

```
if (IK(inst) == IK_MOV && IM(inst) == M_Cr && IK(prev) != IK_CMOVCC
      && (IM(prev) == M_CI || IM(prev) == M_Cr || IM(prev) == M_CM)
      && IREG(prev) == IREG2(inst) && use_cnt[IREG(prev)] == 1) {
   def_cnt[IREG(prev)]--;
   use_cnt[IREG(prev)]--;
   IREG(prev) = IREG(inst);
   prev->next = inst->next;
   inst->next->prev = prev;
   prev->next = inst->next;
   inst = prev;
   continue;
}
```

The pattern essentially checks whether the physically first out of two instructions (`prev`) writes into a register and the next instruction (`inst`) copies the register to another register. Then, if the original register has only one use (the second instruction) the copy is not necessary, and the move can be realized directly. Other than the change in register, the rule deletes the redundant copy, and marks the one remaining instruction as current for the next investigation. Use and definition counts are updated accordingly.

This pattern cover surprising lot of cases, for example it can delete copies in all of the following:

```
lea t32, [rbp-16]
mov t14, t32

mov t27, 1
mov t18, t27
```

As this optimization removes an instruction it allows more optimizations that would otherwise be outside of reach. Despite being a simple form of copy propagation it works so well, because a lot of copies are present in translation to x86-64's two address code, many of them can be fused together with constant materialization.

4.8.6.2 Duplication of patterns

Even though use-def based optimization can propagate constants or memory locations, they are currently only tried last. Less general local peephole optimization patterns are tried first. For example, with the proper use and def count checks a pattern which transforms the following:

```
mov t22, [H]
mov t23, [...]
add t23, t22
```

to:

```
mov t23, ...
add t23, [H]
```

performs much better than the "data-flow" version from section 4.8.3 based on the `try_combine_memory` function. This is purely because it is able to bring the peephole window back to the first instruction, which cascades to many other optimizations.

Improvements to allow the use-def based optimizations to go back would be necessary to prevent duplication in patterns, while keeping the quality of the generated code. Implementing such improvement is more tricky than it might seem, because there are performance as well as clarity concerns.

4.8.6.3 Avoiding dead ends

As mentioned in section 4.8.2, we prefer the `lea` instruction for arithmetic, because it allows subsequent optimizations to fold more arithmetic into a single instruction. Again, these optimizations cascade well, and we can even turn C code like this:

```
int f(int *arr, int a, int b, int c) {
    return arr[c] = arr[a] + arr[b];
}
```

Into the following (without the uninteresting prologue and epilogue):

```
mov rax, [rdi+8*rsi]
add rax, [rdi+8*rdx]
mov [rdi+8*rcx], rax
```

This stems from the great composability of the `lea` instruction. To not miss these optimizations our peephole optimizer prefers `lea` even for situations where other instructions might be better, e.g. it can generate the following `lea`s:

```
lea t12, [t12+t13]
lea t12, [t12+1]
lea t12, [t12-1]
```

The first instruction could be replaced by a single `add` instruction. On some low-end Intel microarchitectures [Intel Optimization Manual, 2023], the `add` is recommended, because the processors use separate Address Generation Unit (AGU) for `lea` calculations which means doing normal arithmetic there imposes additional synchronization.

The last two instructions are replaceable by `inc` and `dec` respectively. These have their own problems with partial writes to flags (see section 4.8.2), but there are no big penalties on recent processors.

As using `lea` is generally not worse in any significant way (and certainly not so on recent microarchitectures [Intel Optimization Manual, 2023]), we believe it is okay to leave the situation as is.

4.8.6.4 Optimization order

Because we try to match and apply peephole optimization patterns one by one, we want to pay attention to their *order*, because in some cases optimizations may prevent other ones.

As an example, take calculation of struct field's address. It is formed by adding the offset to the field to the address of the struct itself. TinyC code like this:

```
struct S {
    int a;
    int b;
};
S a;
int global_offset() {
    return a.b;
}
```

which will be expanded to something like this:

```
lea t25, [a]
mov t26, 8
mov t17, t25
add t17, t26
mov t18, [t17]
mov rax, t18
```

While it could be simplified to a single instruction:

```
mov rax, [a+8]
```

Our optimizer is able to do this transformation through a series of steps, most of which we have already shown. But at some point we get to:

```
lea t17, [a]
add t17, 8
mov t18, [t17]
```

The addition of constant can be folded into the displacement of the memory specification:

```
lea t17, [a+8]
mov t18, [t17]
```

Note that here we have RIP-relative addressing (see section 4.3.2.2 for more information about labels). There are two nice things about this particular transformation: it reduces the number of definitions, and if the `lea` instruction is the last remaining definition of `t17`, we can mark it as the *only* definition of `t17`. Importantly, this makes the `[a+8]` address calculation foldable later due to use-def based peephole patterns (see section 4.8.3). In this specific example, use-def based optimization is not necessary, because the load instruction is next to the address calculation.

But, if we modify the example slightly:

```
void local_offset() {
    S a;
    S b;
    [...]
    a.a = b.a;
}
```

we can get into a similar situation with regards to the offset calculation, where from this:

```
lea t29, [rbp-16]
mov t30, 0
mov t19, t29
add t19, t30
lea t31, [rbp-32]
mov t32, 0
mov t20, t31
add t20, t32
mov t21, [t20]
mov [t19], t21
```

it is not a problem to get to this (if we peepholed each address calculation separately):

```
lea t19, [rbp-16]
add t19, 0
lea t20, [rbp-32]
add t20, 0
mov t21, [t20]
mov [t19], t21
```

Here, if like explained above we fold the `add` to the displacement of the `lea` and mark `t19` and `t20` as the only definitions, we can fold them into the loads and stores below:

```
mov t21, [rbp-32]
mov [rbp-16], t21
```

However, there is a simpler pattern that can make us fail to do that. The `add` instruction has no observable flags and adds zero, so it may simply get deleted. This is however even more local, and doesn't notice that the `lea` before it has now become the last definition, leaving the `lea` instructions unfoldable:[1]

```
lea t19, [rbp-16]
lea t20, [rbp-32]
mov t21, [t20]
mov [t19], t21
```

We might make the use-def optimizations more robust by using full use-def and def-use chains (see section 3.4, where they are mentioned) or in this case, simply guarantee that the pattern matching two instructions is tried before the single pattern instruction. Intuitively longer or in general *more specific* patterns should always be preferred and as mentioned, local patterns should be preferred to use-def ones (section 4.8.3).

4.9 Register allocation

Register allocation by graph coloring and in particular the iterated register coalescing [George et al., 1996] algorithm we implement, does not consist of one simple step, but of multiple. Additionally, the algorithm is repeated until it succeeds. The basic means of operation of the algorithm are briefly introduced here:

[1] Actually the load instruction can get folded, because it is next to the address calculation, but the store address will certainly not be folded.

1. *Liveness analysis.* First we need to analyze where are virtual registers *live*.
2. *Build interference graph.* From the liveness information we construct the interference graph.
3. *Calculate spill costs.* We calculate how costly would spill of each virtual register be as well as mark some registers unspillable.
4. *Perform iteration of iterated register coalescing graph coloring.* This is the initialization and main loop of the [George et al., 1996] algorithm. It runs until the graph is fully simplified, which also determines a coloring order.
5. *Assign registers.* Virtual registers are colored one by one according to the order and coalescing determined in the previous step. The previous step always succeeds. Virtual registers are at most marked as *potential spills* and may still be assigned color. If no color is left for some of them in this step, they are an *actual spills* and this step returns the list of virtual registers to spill.
6. *Rewrite the program.* If the previous step failed the program is rewritten to include loads and stores of spilled virtual registers and all the above steps are attempted again.
7. *Apply register assignment.* If the graph coloring was successful (i.e. there were no *actual spills*), then the program can be rewritten to change occurrences of virtual register to their assigned physical registers.

The specialty of the iterated register coalescing algorithm lies mainly in step 4, but it has also consequences on previous steps, notably on the design of data structures (discussed in section 4.9.6).

The overall design of the register allocator in our compiler keeps it as a separate, opaque component. We store all data needed for the register allocator in a single struct called `RegAllocState`. The user of the register allocator obtains the state with a call to a *create* function and frees the state with a *free* function. Register allocation itself requires the allocated state and a machine function for which to do the register allocation. The register allocator returns the function modified to not use any virtual register, which sometimes means that additional spill code is added. The API design allows a single state to be reused for register allocation of many functions and thus greatly reducing the costs of allocating the state.

```
RegAllocState *reg_alloc_state_create(Arena *arena);
void reg_alloc_state_free(RegAllocState *ras);

void reg_alloc_function(RegAllocState *ras, MFunction *mfunction);
```

Most of the register allocator is target independent. The only x86-64 dependent things are currently the routines for creating loads and stores from stack, allocation of stack slots and the number of physical registers as well as the descriptor tables (see section 4.3.2 for more details about the back end representation).

More information about the steps, as well as design and implementation considerations are the subject of the following subsections.

4.9.1 Liveness analysis

For liveness analysis we use the classic [Kam et al., 1977] approach of iterative data-flow analysis:

1. We construct data-flow equations.
2. We solve them with an iterative algorithm.

In liveness we propagate information in control flow from future to past—each *use* means that a virtual register starts being *live* and each *definition* means that a virtual register stops being *live*. Concrete data-flow equations that capture this are given for example by [Appel et al., 1998]:

$$in[n] = use[n] \cup (out[n] - def[n]) \tag{1}$$
$$out[n] = \bigcup_{s \in succ[n]} in[s] \tag{2}$$

The formulation is based on *live-in* and *live-out* sets, which capture the state of liveness along control flow edges. Liveness information in a control flow node is given by the liveness in all *successors*, i.e. we propagate information from the future to past.

Iterative data-flow analysis starts with all *live-in* and *live-out* sets empty, and refines them until reaching a fixed point (i.e. the point where all the sets stabilize and are equal to those in previous iteration). This can be quite expensive and can require quite a lot of memory in a naive solution, so in our implementation we perform a few optimizations:

- Our control flow edges are *basic blocks*. We only keep liveness sets for the edges to and from basic blocks, which requires much less memory than doing so for each *instruction*. To compute basic block's live-out from the live-in of all successors equation (2) suffices. To compute live-in of a block from its live-out we need to iterate over the basic block *backwards* and use equation (1) for each instruction.
- We don't store in memory all live-in and live-out sets, but only live-in sets of all basic blocks. We can recompute live-out set for any basic blocks according to equation (2).
- We use a *work list* based approach [Cooper et al., 2006] in which we first insert all blocks into a work list, then as long as the work list is not empty, we keep removing a block from it, recompute its liveness sets and insert its *predecessors* back to the work list. This means that we iterate over each block once in the beginning, and then only on demand as we find changes in block's *successors*.
- Blocks use information from their *successors*, which means that if we processed all blocks *after* their successors, we could find the solution in a single iteration. In practice, control flow graphs contain loops, so processing all successors of a block before the block is not possible, but we can use a block order which tries to put block's successors before it, such as the postorder of the control flow graph, which we already have available (see section 4.3.1.5).

In the actual implementation this becomes the following:

```
void liveness_analysis(RegAllocState *ras) {
   MFunction *mfunction = ras->mfunction;
   WorkList *live_set = &ras->live_set;

   wl_init_all_reverse(&ras->block_work_list, mfunction->mblock_cnt);
   Oper b;
   while (wl_take(&ras->block_work_list, &b)) {
      MBlock *mblock = mfunction->mblocks[b];
      Block *block = mblock->block;
      get_live_out(ras, block, live_set);

      Inst *inst = mblock->insts.prev;
      while (inst != &mblock->insts) {
            live_step(live_set, mfunction, inst);
            inst = inst->prev;
        }

        if (!wl_eq(live_set, &ras->live_in[b])) {
            WorkList tmp = ras->live_in[b];
            ras->live_in[b] = *live_set;
            *live_set = tmp;
            FOR_EACH_BLOCK_PRED(block, pred)
                wl_add(&ras->block_work_list, (*pred)->mblock->index);
        }
    }
}
```

The same data structure (`WorkList`) is used for representing both the work list for blocks that need to be processed as well as the *liveness sets*—sets need to support fast unique addition as well as removal and iteration and clearing, which is pretty similar to work list's needs. Our work list implementation (section 4.3.3) supports all these operations. As mentioned live-out sets are not stored, `RegAllocState` only holds live-in sets:

```
struct RegAllocState {
    [...]
    WorkList block_work_list;
    WorkList live_set;
    WorkList *live_in; // WorkList for each block
    [...]
}
```

Block's indices correspond to their positions in `mfunction->mblocks`, which lists them in *reverse postorder*. These indices are also used for subscripting the `live_in` and other arrays. The arrays, sets and work lists in `RegAllocState` are allocated with enough capacity for all blocks or all virtual registers (depending on the use).

```
void get_live_out(RegAllocState *ras, Block *block, WorkList *live_set) {
    wl_reset(live_set);
    FOR_EACH_BLOCK_SUCC(block, succ)
        wl_union(live_set, &ras->live_in[(*succ)->mblock->index]);
}
```

The live step function mostly implements equation (1) for an instruction:

```
void live_step(WorkList *live_set, MFunction *mfunction, Inst *inst) {
    // Remove definitions from live.
    for_each_def(inst, remove_from_set, live_set);
    // Add uses to live.
    for_each_use(inst, add_to_set, live_set);
}
```

Simple iteration over register uses and definitions was one of the motivations for our representation for instructions (see section 4.3.2) and it works out nicely, since we can use generic callback-based iterators for iterating over all definitions and uses.

After the function `liveness_analysis` runs, `live_in` sets for each block are calculated. From these we can derive the set of live virtual registers at any program point by starting in the right basic block, computing live-out, and iterating backwards to the program point of interest updating the live-set with `live_step`, similarly as we do in the liveness analysis itself.

4.9.2 Build interference graph

Now that we know which virtual registers are live at different program points, we use them to construct the *interference graph*.

Formally, two virtual registers interfere when they are live at the same time (definition 3.2). We can find the live-set for each program point by iterating over all blocks (in any order) and instructions (backwards) maintaining a live-set with the `live_step` function. But adding interference between all ℓ live in the live-set would mean adding $\ell \cdot (\ell - 1)$ at each program point—this is both expensive and adds too many edges repeatedly if virtual registers are alive for longer periods of times.

Instead Chaitin [Chaitin et al., 1981] suggests a more practical notion of interference:

Definition 4.1 Interference. Two virtual registers *interfere* if one of them is live at the definition point of other. [Chaitin et al., 1981]

This definition translates much better to an implementation, because this means at each instruction adding interference for all definitions in that instruction with all ℓ members of live-set (thus for most instruction this is ℓ edges, in general $c \cdot \ell$, where c is a constant).

The actual implementation of building the interference graph is very similar to performing liveness analysis (section 4.9.1), except that we can iterate over each block just once, and call the `interference_step` function (shown below) instead of `live_step`—apart from updating the liveness instruction to instruction, we need to add the interference edges:

```
void interference_step(RegAllocState *ras, WorkList *live_set, Inst *inst) {
   if (is_move(inst)) {
      for_each_use(inst, remove_from_set, live_set);
      add_move(ras, inst);
   }

   for_each_def(inst, add_to_set, live_set);

   FOR_EACH_WL_INDEX(live_set, j) {
      for_each_def(inst, add_interference_with, live_set->dense[j]);
   }

   for_each_def(inst, remove_from_set, live_set);
   for_each_use(inst, add_to_set, live_set);
}
```

The most important part above is the loop which for each member of `live_set` adds interference with each definition. But before doing that, we actually add all *definitions* to the live set. By adding definitions to the live-set and then adding interferences of all definitions with all live, we also add interferences across all *defined registers*, which is important, to prevent assignment of the same physical register to any two definitions in the same instruction. Adding the definitions to the live set is not really correct in the iteration, since we ought to add *uses*, not *definitions*. But when stepping the liveness for an instruction, we actually remove definitions from the live set, *before* adding the uses, so the fact that we add the definitions to the live set temporarily is not observable from the outside. The liveness step is realized by the last two calls, which are the same as in `live_step`.

Before any of that, we also have special handling of copy instructions. For example an instruction like

```
mov t19, t20
```

shouldn't add an artificial interference between the two virtual registers, because then they *can't* be coalesced (recall section 3.8.4.3, we can only coalesce non-interfering virtual registers). Other instructions can find that the virtual registers are live at the same time at a *different program point*, but the copy instruction itself shouldn't add the interference. This can be done nicely, by removing the used register (i.e. `t20`) from the live-set—then it won't be present when interferences of definitions and live-set members are added.

Last, but not least, we also call `add_move` function to accumulate all copy instructions, which is important for the later coalescing stage.

4.9.2.1 Calling conventions

In sections 3.8.3.6 and 3.8.4.3 we claimed that the combination of live range splitting and interferences added through right definitions and uses can be used to model calling conventions passing arguments or return values in physical registers (which is done on all commonly used x86-64 calling conventions, see section 2.5). In section 4.7.3 we described how the live range splitting of callee saved registers is handled. But so far we haven't explained how the interferences are added in our implementation, since the instruction representation (see section 4.3.2) only holds the registers explicitly listed by the instruction and there is no space for any calling convention related registers and their uses and definitions.

Before describing the implementation details, let's show on an example how exactly we want to model uses and definitions on the following source program:

```
int f(int c) {
   return g() + h() + c;
}
```

After peephole optimization, right before register allocation:

```
f:
.L0:
        ; entry
        push rbp
        mov rbp, rsp
        sub rsp, 42

        mov t28, rbx
        mov t29, r12
        mov t30, r13
        mov t31, r14
        mov t32, r15

        mov t18, rdi

        call g
        mov t21, rax

        mov rdi, t18
        call h
        lea t26, [t21+rax]
        add t26, t18
        mov rax, t26

        mov rbx, t28
        mov r12, t29
        mov r13, t30
        mov r14, t31
        mov r15, t32

        mov rsp, rbp
        pop rbp
        ret
```

We see the classic prologue and epilogue creating and destroying the stack frame and since we don't yet know the final stack space required (there might be spills) a dummy amount is listed and will be fixed up much later. Then there are the live range splits for the five callee saved registers (`rbx` and `r12` through `r15`). There is also a live range split for the argument `c`, which is passed in register `rdi`, but saved into `r18`. Another live range split is present for the return value of function `g`, which is moved from `rax` to `t21`. The argument for `h` is also split and passed in `rdi`. A copy of the return value of `h` has actually been optimized out and `rax` is used directly in the following addition (`lea` instruction). After both calls all three values are added in two instructions and the result is moved to `rax`, the return value register.

The `call` instruction has only one argument—the label of the function to call. It doesn't in any way communicate explicitly, that the caller saved registers (like `rdi` or `rsi` which are also used for passing arguments) are not preserved throughout the call. There is also no explicit definition of the callee saved and argument registers, even though we use them in the function. As established mainly in section 3.8.3.6, we want to add:

- Uses of actually used *argument registers* to the `call` instruction. Otherwise the definitions which assign parameters to registers would see no uses and thus in the backwards computation of liveness they wouldn't ever be *added* to the live-set. On the other hand, we need to add uses of only the registers actually used for argument passing—otherwise they would be added to the live-set unnecessarily and without any definitions—meaning a lot of interferences would be added and use of the argument registers would effectively be forbidden.
- Definitions of all *caller saved registers* to the `call` instruction. Caller saved registers are not preserved by the function call, anything live across the call has to either be saved in a callee saved register (because every register is either caller or callee saved) or spilled to memory.
- Uses of all *callee saved registers* to the `ret` instruction. The caller expects the callee saved registers to be preserved, thus modelling uses of them in the `ret` instruction adds them to the live set for as long as they are not defined. If we defined them in the entry to the function we would have effectively added interferences of them with everything else in the function and thus preserved them correctly, but not allowed them to be allocated.

  But in our case, they are split, and their definitions appear right before the epilogue and thus they are kept in the live set only for a short time. It is still important that we model the uses in the `ret` instruction though—it adds interferences to the registers holding the callee saved register values. These interferences are important for graph coloring algorithm, as precise degrees of nodes are expected.
- Definitions of *argument and callee saved registers* in the entry point. They are "passed" by the caller. Adding the definitions isn't strictly needed as the definitions won't prevent any additional interferences. But they assure that all registers

have at least one definition, which makes it consistent for the purposes of peephole optimization (where we track the uses an definitions of registers, see section 4.8.3).

Adding all the uses and definitions as additional *operands* to the `Inst` structure (introduced in section 4.3.2), is a bit wasteful, considering we want to attach the extra defined and used registers mostly statically to instructions kinds, except sometimes we want to limit the number of registers to *argument count*.

Because of this regularity, we opted to attach the additional uses and definitions to instruction *modes*. The existing modes `M_LCALL` (call label) `M_rCALL` (indirect call through register) and `M_RET` (return instruction) can be extended with pointers to these additional definitions and uses.

A bit more problematic is attaching definitions and uses to the entry point—there is no explicit instruction there, and we don't want to handle neither function entry nor exit points specially. In the end we chose to introduce an additional instruction kind `IK_ENTRY` with a mode `M_ENTRY` whose sole purpose is to hold the additional definitions and uses. The instruction is otherwise mostly skipped as a no-op instruction. It is present even in the listing above—it's text printout is "`; entry`" (i.e. it is a comment).

Code-wise, the definition of the `ModeDescriptor` type is enhanced with a couple of fields related to these additional definitions and uses:

```
typedef struct {
   [...]
   bool use_cnt_given_by_arg_cnt;
   bool def_cnt_given_by_arg_cnt;
   Oper *extra_defs;
   Oper *extra_uses;
} ModeDescriptor;
```

To avoid having to specify the number of physical registers in these lists, they are terminated with the special register `R_NONE`. If the number of extra definitions or uses is determined by the argument count (as is for the number of either received arguments for the entry instruction or passed arguments in case of call instruction), it is signified in the mode descriptor by a boolean flag, and the count is read using the macro `IARG_CNT` from one of the operand fields. These additional definitions and uses are also used for long division, where `rax` and `rdx` are both read and written implicitly, while there is an extra register or memory argument specifying the divisor, because listing `rax` and `rdx` wouldn't fit into the compact representation of operands of x86-64 instructions. Again, similarly to real encoding of x86-64 instructions, where the use of `rax` and `rdx` is only implicit.

In practice, the mode descriptors are extended like this:

```
#define IARG_CNT(inst) ((inst)->ops[5])

Oper none[] = { R_NONE };

Oper rax_rdx[]       = { R_RAX, R_RDX, R_NONE };
Oper caller_saved[]  = { R_RAX, R_RCX, R_RDX, R_RSI, R_RDI,
                         R_8, R_9, R_10, R_11, R_NONE };
Oper argument_regs[] = { R_RDI, R_RSI, R_RDX, R_RCX, R_8, R_9, R_NONE };

ModeDescriptor mode_descs[] = {
   [M_Rr]    = { [...],  0, 0, none, none },
   [M_rCALL] = { [...],  1, 0, caller_saved, argument_regs },
   [M_ADr]   = { [...],  0, 0, rax_rdx, rax_rdx },
};
```

And the iteration over all uses look like this:

```
void
for_each_use(Inst *inst,
             void (*fun)(void *user_data, Oper *use),
             void *user_data)
{
   ModeDescriptor *mode = &mode_descs[inst->mode];
   for (size_t i = mode->use_start; i < mode->use_end; i++)
      fun(user_data, &inst->ops[i]);

   if (mode->use_cnt_given_by_arg_cnt) {
      size_t use_cnt = IARG_CNT(inst);
      for (size_t i = 0; i < use_cnt; i++)
         fun(user_data, &mode->extra_uses[i]);
   } else {
      for (Oper *use = mode->extra_uses; *use != R_NONE; use++)
         fun(user_data, use);
   }
}
```

The callback-based iterator passes pointers ("references") to the registers. This is needed for spilling, where we rewrite uses and definitions.

Perhaps greater flexibility could be achieved by not keeping the compact encoding and instead representing the definitions and uses in a more expensive way (i.e. with each instruction, not mode). But the compact operand packing, close to x86-64 instructions is one of the reasons why peephole optimization can be done so expressively even with hand-coded rules.

The approach is mostly target independent—each target can have different mode descriptors. The macro `IARG_CNT` is x86-64 specific, but extending the approach to have a per-target operand index for the argument count would be straightforward.

With all the implementation details in place, our example above register allocates and peephole optimizes successfully to the following:

```
f:
.L0:
        push rbp
        mov rbp, rsp
        sub rsp, 16
        mov [rbp-16], rbx
        mov [rbp-8], r12
        mov r12, rdi
        call g
        mov rbx, rax
        mov rdi, r12
        call h
        lea rax, [rbx+rax]
        add rax, r12
        mov rbx, [rbp-16]
        mov r12, [rbp-8]
        mov rsp, rbp
        pop rbp
        ret
```

The “entry instruction” is optimized out, since it is no longer needed after register allocation. Callee saved registers `rbx` and `r12` are used for values live across call, and as their virtual registers got spilled, the former values in callee saved registers got automatically saved in stack slots. Other than for the other callee saved registers, coalescing wasn’t applicable much—the registers were mostly restricted by the calling convention or interfered with each other due to being live at the same time.

4.9.3 Iterated register coalescing

A lot of other graph coloring register allocation approaches could reuse many parts of our implementation. However, the *main simplification loop* which we describe in this section is the main characteristic of the iterated register coalescing (“IRC”) algorithm by George and Appel [George et al., 1996], which we also describe in context of other graph coloring register allocators in section 3.8.4.3. As we will be working mainly on an interference graph of virtual registers, we will also be calling virtual registers *nodes* of the interference graph, and we will call interferences *edges*. The ideas and implementation of the algorithm in our implementation are based also on pseudo code listed in [George et al., 1996], though we chose more descriptive names, and additionally include clarifications and corrections to the algorithm.

The main loop of IRC interleaves *simplification* and *conservative coalescing*. This leads to more coalescing, while still being as safe as earlier approach. The main loop’s purpose apart from the coalescing is to push all the nodes on to a stack, even the potential spills, because we use optimistic coloring. The actual coloring of the graph is handled in a later stage, which is common to other approaches like Briggs’ and uses the order imposed by the stack for coloring the registers.

For a start, let’s first consider the simplification phase of Briggs’ algorithm, which also does optimistic coloring, but contrary to IRC does all the coalescing before starting with simplification. The simplifications are based on Chaitin’s heuristic, which says that every node with less than k neighbours (where k is the number of available registers) can be pushed onto the stack and “removed from the graph”—later when coloring the node will be trivially colorable when popped from the stack and “reinserted back into the graph”. Based on this heuristic we want to partition the nodes not yet pushed onto the stack into two different categories:

1. Low ($< k$) degree nodes. Sometimes also called “insignificant” or “trivially colorable”.
2. High ($\geq k$) degree nodes. Sometimes also called “significant”.

Strict separation of these categories allows our efforts to concentrate on the low degree nodes—those can be removed from the graph as they already fulfill the condition imposed by Chaitin’s heuristic. Of course, their removal decreases degrees of neighboring nodes, and occasionally high degree node becomes low degree and we need to move it from the category to the first. Partitioning the nodes into two categories initially is easy. Then, as long as the set of low degree nodes is not empty, the algorithm can simplify the interference graph and keep constructing the coloring order on the stack.

But when the set of low degree nodes becomes empty, and there are still nodes left in the high degree set, then we need to choose a node to be a *potential spill* and push it to the stack and remove it from the graph despite having a high degree. This either allows other nodes to become low degree or requires even more potential spills. Since the “remove from graph and push onto the stack” action is already performed on every node in the low degree set: we can just move the potential spill to the low degree set and let it be handled by the same simplification mechanism. Consequently we can find more appropriate names for the two sets:

- *Simplify set.* Contains nodes intended for simplification.
- *Spill set.* Nodes that are currently spill candidates, because their degree is high. They can move to the simplify set either when their degree becomes low or they are chosen as potential spill.

We can represent the two categories of nodes with two *work lists*—we already use work lists for representing other sparse sets (like the live-set in section 4.9.1), because our work list implementation (section 4.3.3) supports set operations efficiently. Our work list can even be used for the stack, which gives us the following main simplification loop:

```
while (true) {
   while (wl_take_back(&ras->simplify_wl, &i)) {
      simplify_one(ras, i);
   }
   if (wl_empty(&ras->spill_wl)) {
      break;
   }
   choose_and_spill_one(ras);
}
```

The loop always tries to remove nodes from the simplify work list as long as possible. If no node is available for simplification, a node from the spill work list is spilled. If the spill work list is empty, it means that we have already fallen-through the simplify loop and both work lists are empty, thus we can break out of the loop. If we for now ignore the choice of the best spilled candidate, even the helper functions are straightforward:

```
void simplify_one(RegAllocState *ras, Oper i) {
   wl_add(&ras->stack, i);
   for_each_adjacent(ras, i, decrement_degree);
}

void decrement_degree(RegAllocState *ras, Oper u) {
   if (ras->degree[u]-- == ras->reg_avail) {
      wl_remove(&ras->spill_wl, u);
      wl_add(&ras->simplify_wl, u);
   }
}

void choose_and_spill_one(RegAllocState *ras) {
   Oper candidate = ⟨the best spill candidate⟩;
   wl_remove(&ras->spill_wl, candidate);
   wl_add(&ras->simplify_wl, candidate);
}
```

Simplifications and spills move nodes from the spill work list to simplify work list. In the case of simplifications, nodes are moved when their degree decreases below k (represented by `ras->reg_avail`). The simplified nodes themselves are pushed onto a stack. The interference graph itself is not modified in any explicit way—with the push of a node onto the stack we remove it from the graph *implicitly*—until the node is popped back, we have to presume it and its edges removed from the graph. Due to that, the `for_each_adjacent` function skips nodes which are present on the stack and we maintain a degree of each node to detect simplifications without having to check presence of neighbours on the stack.

The simplifications in the iterated register coalescing are still based on the same Chaitin heuristic, so the same simplification mechanism applies. But, because we want to interleave simplifications with coalescing, we need to obey one limitation: we can't coalesce any node pushed onto the stack, because it is *removed* from the graph.

To not miss any coalescing opportunities, we have to try all coalescings of a node before we push it (i.e. remove it from the graph). We only remove from the graph nodes put on to the simplify work list, and to prevent putting coalescable nodes onto the simplify work list, we introduce a third category of nodes: low degree move-related nodes—nodes that have a low degree, but are either a source or destination in at least one copy (move) instruction. We than add to the simplify work list only those low degree nodes that are not move-related, i.e. those which can't be coalesced. Though like with high degree nodes, if we can't find opportunities for simplifications, we have to give up, and move something to the simplify work list. Instead of choosing a node from the spill work list, it is preferable to give up any coalescing of a move-related node. We will call giving up on coalescing *freezing* and like with spills, separate nodes which we can freeze into a separate work list:

- *Simplify work list.* Contains nodes intended for simplification, i.e. removal from graph and push onto the stack.
- *Freeze work list.* Nodes that are currently freeze candidates, because they are low degree and move-related. They can move to the simplify work list either when they stop being move-related (i.e. all their moves are either coalesced or given up individually) or when they are chosen to be frozen (i.e. all their moves are given up).
- *Spill work list.* Nodes that are currently spill candidates, because their degree is high. They can move to the freeze work list when their degree becomes low and they are move-related, or they can move to simplify work list if their degree becomes low and they are not move-related. They can also move to the simplify work list if they are chosen as potential spill.

We also need to distinguish several kinds of moves:

- Active moves. Moves that we want to actively check for coalescing.
- Inactive moves. Moves that we already checked for coalescing, but were declined by the conservative heuristic. If there is a chance that a change might make the conservative heuristic allow the coalescing, an inactive move should be made active and the coalescing tried again.
- Given up (*frozen*) moves. Moves that are no longer coalescable in any capacity. Either because the source and destination interfere or we have given up on them by freezing the source or destination.

We also partition the moves into work lists. Representing frozen moves is actually not needed, since they can be detected by not being present on the other two work lists.

For the high level algorithm, we will revisit Briggs' simplification. There we would simplify in a loop, then fall-through to spill and if any spill was made, we would go back to simplification, otherwise we would end the loop. This can be also written like this in C:

```
simplify:
while (wl_take_back(&ras->simplify_wl, &i)) {
   simplify_one(ras, i);
}
if (!wl_empty(&ras->spill_wl)) {
   choose_and_spill_one(ras);
   goto simplify;
}
```

The falling-through is made implicit and the backward jumps are made explicit through `goto`. With IRC we have more chances to add to simplify work list other than spilling:

- Coalescing two nodes into one decreases degrees of nodes that were interfering with both. This means that they can be added to the simplify work list through `decrement_degree`.
- Before resorting to spilling, we want to freeze a node if any can be frozen. Giving up moves (i.e. keeping the move instructions) is better than spilling (i.e. adding loads and stores).

We can thus introduce processing of active moves and freezing, with more explicit backward jumps to simplification:

```
simplify:
while (wl_take_back(&ras->simplify_wl, &i)) {
   simplify_one(ras, i);
}
if (wl_take(&ras->active_moves_wl, &i)) {
   coalesce_move(ras, i);
   goto simplify;
}
if (wl_take_back(&ras->freeze_wl, &i)) {
   freeze_one(ras, i);
   goto simplify;
}
if (!wl_empty(&ras->spill_wl)) {
   choose_and_spill_one(ras);
   goto simplify;
}
```

This code captures the idea of IRC very well: first simplify as much as possible, then try coalescing, and go back to simplification. Since full simplification (removal of all nodes from the graph) is our end goal, if the simplification work list ever becomes empty with the graph still being non-empty, we add even coalescable or high degree nodes to the simplify work list.

More guided freeze choice using *free costs* (analogous to *spill costs*) is possible [Leung et al., 1998], and would be a nice future improvement to our implementation.

The tricky part of the IRC algorithm is getting the *transitions* right. Initially, putting the nodes and moves into right work lists is straightforward:

```
void initialize_worklists(RegAllocState *ras) {
   wl_init_all(&ras->active_moves_wl, move_cnt);
   wl_reset(&ras->inactive_moves_wl);

   size_t vreg_cnt = ras->mfunction->vreg_cnt;
   for (Oper u = ras->first_vreg; u < vreg_cnt; u++) {
      if (is_significant(ras, u)) {
         wl_add(&ras->spill_wl, u);
      } else if (is_move_related(ras, u)) {
         wl_add(&ras->freeze_wl, u);
      } else {
         wl_add(&ras->simplify_wl, u);
      }
   }
}
```

But any following change in degree or moves may require a transition of node or move into a different work list. Notably transitions which previously put nodes right into simplify work list now need to check whether the node is move-related. One such place is the `decrement_degree` function:

```
void decrement_degree(RegAllocState *ras, Oper u) {
   if (ras->degree[u]-- == ras->reg_avail) {
      enable_moves_for_one(ras, u);
      for_each_adjacent(ras, u, enable_moves_for_one);
      wl_remove(&ras->spill_wl, u);
      if (is_move_related(ras, u)) {
         wl_add(&ras->freeze_wl, u);
      } else {
         wl_add(&ras->simplify_wl, u);
      }
   }
}
```

Apart from moving the node from spill work list to either freeze or simplify work list, moves of the node and its neighbours are enabled (moved from inactive to active). This is because Briggs' heuristic for coalescing is based on number of *significant neighbours*—a node becoming insignificant *may* make moves of all neighbours coalescable even if they were previously denied as potentially unsafe.

Enabling moves is a straightforward transition from inactive moves to active moves. This is because both active and inactive moves count towards the node being move related and there is no change in status of any node.

```
void enable_move(RegAllocState *ras, Oper u, Oper m, Inst *move) {
   (void) u;
   if (wl_remove(&ras->inactive_moves_wl, m)) {
      wl_add(&ras->active_moves_wl, m);
   }
}

void enable_moves_for_one(RegAllocState *ras, Oper u) {
   for_each_move(ras, u, enable_move);
}
```

On the other hand, when nodes move straight to simplify work list, either because of spilling or freezing, all their moves are given up entirely, i.e. they will be removed from both inactive and active moves and not considered further:

```
void choose_and_spill_one(RegAllocState *ras) {
   Oper candidate = <the best spill candidate>;
   wl_remove(&ras->spill_wl, candidate);
   wl_add(&ras->simplify_wl, candidate);
   freeze_moves(ras, candidate);
}

void freeze_one(RegAllocState *ras, Oper i) {
   wl_add(&ras->simplify_wl, i);
   freeze_moves(ras, i);
}
```

As opposed to enabling (inactive $\rightarrow$ active), freezing a move (giving up on the move entirely) can change the status of the node—the removed move can be the last one related to the node and the node can possibly transition from the freeze work list to the simplify work list, though it stays in spill work list if it is high degree:

```
void freeze_move(RegAllocState *ras, Oper u, Oper m, Inst *move) {
   if (!wl_remove(&ras->inactive_moves_wl, m)) {
      wl_remove(&ras->active_moves_wl, m);
   }
   Oper op1 = get_alias(ras, move->ops[0]);
   Oper op2 = get_alias(ras, move->ops[1]);
   Oper v = op1 != u ? op1 : op2;
   if (!is_move_related(ras, v) && !is_significant(ras, v)) {
      wl_remove(&ras->freeze_wl, v);
      wl_add(&ras->simplify_wl, v);
   }
}

void freeze_moves(RegAllocState *ras, Oper u) {
   for_each_move(ras, u, freeze_move);
}
```

There are two subtle details in the `freeze_move` function. First, the operands of a move in the move instruction are listed as the original nodes, but coalescing might have merged them into other nodes, so a resolution through aliases is needed. Next, when freezing moves of `u` we don't need to consider `u` transitioning from freeze work list to simplify work list—we are already removing all the moves. But we need to do the check for the other move operand, `v`.

4.9.3.1 Coalescing

The active moves work list holds the indices into an array listing all move instructions. When considering the operands of an active move instruction for coalescing, the function `coalesce_move` is called, where there are 4 cases:

1. The registers were already coalesced. The move needs to be given up, *frozen* to not be investigated again.
2. The registers are interfering and can't be coalesced. The move also needs to be *frozen* to not be investigated again, since the there is no chance of ever coalescing.
3. The registers can be coalesced and the coalescing heuristic allows the coalescing. Then we also want to not investigate the move again (i.e. remove the move from work lists), but also need to merge the two nodes into one.
4. The registers can be coalesced and the coalescing heuristic *doesn't* allow the coalescing. Then we transition the move from active moves to inactive moves and will try it again in case the situation improves (the move is enabled) or the move is frozen for other reason.

In an implementation this looks like this:

```
void coalesce_move(RegAllocState *ras, Oper m) {
   Inst *move = garena_array(&ras->gmoves, Inst *)[m];

   Oper u = get_alias(ras, move->ops[0]);
   Oper v = get_alias(ras, move->ops[1]);
   if (v < u)
      ⟨swap u and v⟩

   if (u == v) {
      decrement_move_cnt(ras, u);
   } else if (are_interfering(ras, u, v)) {
      decrement_move_cnt(ras, u);
      decrement_move_cnt(ras, v);
   } else if (are_coalesceble(ras, u, v)) {
      combine(ras, u, v);
      decrement_move_cnt(ras, u);
   } else {
      wl_add(&ras->inactive_moves_wl, m);
   }
}
```

The move is removed from the active moves work list prior to calling `coalesce_move`, so it doesn't have to be removed there. The calls to `decrement_move_cnt` are important, because in three of the situations the move is either given up or coalesced and not to be considered further. This means that a low degree node might become no longer move related, and can move to the simplify work list:

```
void decrement_move_cnt(RegAllocState *ras, Oper u) {
   if (!is_move_related(ras, u) && !is_significant(ras, u)) {
      wl_remove(&ras->freeze_wl, u);
      wl_add(&ras->simplify_wl, u);
   }
}
```

Nodes are always combined into the lexicographically smaller node (*register*). This is done by copying the moves of that node, as well as all the interferences:

```
void combine(RegAllocState *ras, Oper u, Oper v) {
   if (!wl_remove(&ras->freeze_wl, v)) {
      wl_remove(&ras->spill_wl, v);
   }

   ras->alias[v] = u;

   ⟨add moves of v to u⟩

   ⟨for each adjacent t of v⟩ {
      add_interference(ras, u, t);
      decrement_degree(ras, t);
   }

   if (is_significant(ras, u) && wl_remove(&ras->freeze_wl, u)) {
      wl_add(&ras->spill_wl, u);
   }
}
```

The copy of interferences of `v` is realized by adding interference to `u` for each `v`'s neighbour in the interference graph. However, as we are also conceptually removing `v` from the interference graph, all its neighbours have their degree decremented. When a node is neighbouring with either `v` or `u` this addition of a neighbour and removal of a neighbour is a net zero. Though if a node as adjacent with *both*, then it's degree is truly decremented by one, which, if it is high degree, may lead to moving it to either simplify or freeze work lists. We already can handle this with function `decrement_degree`, so we reuse it now.

The combined node `u` can now have a high degree and may have to be moved to the spill work list. This doesn't mean that it will be spilled—as we use a conservative coalescing heuristic, only coalescings that won't lead to spills are allowed (see section 3.8.4.3). The move into spill work list is only temporary until simplifications decrease the degree of the combined node.

4.9.4 Register assignment

In the previous simplification phase (see section 4.9.3), the interference graph was simplified into an empty graph by gradual simplification. Vertices were removed from the graph one by one and pushed onto *the stack*. In the register assignment stage, we will pop nodes one by one from the stack, reinsert them back into the graph, and find colors for them based on the current interference graph.

For each node we have to find a color, which is not assigned to any of the neighbours. Primarily only nodes with degree lower than the number of available registers were pushed. For such nodes we are guaranteed to a find a color, since even if all the neighbours were assigned different colors, a free color is guaranteed.

However, we are using [Briggs et al., 1994]'s *optimistic coloring* and in the simplification phase push even high degree nodes as *potential spills*. When these are popped, a free color for them is not guaranteed, but it may be found if some of the neighbours get assigned the same colors.

From an implementation's perspective we don't need much additional tools other than those already used in previous sections. Notably, we will use the same mechanism for enumerating all node's neighbours in the interference graph. In particular, as established in section 4.9.3 we represent removals of nodes (and their edges) from the interference graph *implicitly*—nodes pushed to the stack are considered to not be present in the graph, even though the interference graph representation isn't changed in any capacity. In register assignment, we need to do the inverse and reinsert nodes back to the graph and we will also represent this *implicitly* with pops from the stack.

Finding a physical for a virtual register works in three phases:

1. Physical registers used by neighbours are marked as used and not available for this allocation.
2. Moves of the virtual register are checked and if the register of the other operand of a coalescable move is available, then it is chosen for the register.
3. Otherwise first free register is chosen.

The second step tries to do what we could call *last chance coalescing*—even when conservative coalescing failed, we may be able to assign a common physical register. Since the other way of choosing a register is the first one available, last chance coalescing can only improve the allocation.

In the actual implementation, it looks roughly like this:

```
Oper find_register_for(RegAllocState *ras, Oper u) {
   u64 used = 0;
   ⟨for each adjacent adj of u⟩ {
      Oper v = get_alias(ras, adj);
      if (wl_has(&ras->stack, v) || is_to_be_spilled(ras, v)) {
         continue;
      }
      used |= 1U << ras->reg_assignment[v];
   }
   ⟨for each move move of u⟩ {
      Oper op1 = get_alias(ras, move->ops[0]);
      Oper op2 = get_alias(ras, move->ops[1]);
      Oper v = op1 != u ? op1 : op2;
      if (u == v || are_interfering(ras, u, v) || wl_has(&ras->stack, v)
            || is_to_be_spilled(ras, v)) {
         continue;
      }
      Oper reg = ras->reg_assignment[v];
      if ((used & (1U << reg)) == 0) {
         return reg;
      }
   }

   for (size_t reg = 1; reg <= ras->reg_avail; reg++) {
      size_t mask = 1 << reg;
      if ((used & mask) == 0) {
         return reg;
      }
   }

   return R_NONE;
}
```

All neighbours (even those not in the interference graph) are iterated over, aliases are resolved, and the if the node is in the interference graph (not popped yet) and not spilled, then it's assigned register is added to the used registers. To improve run time, the set of used registers is packed into a single 64-bit integer and bitwise operations are used—every bit represents a single physical register. When skipping not yet allocated coalescing candidates, we have to be careful about also skipping the self copies (i.e. `mov t12, t12`) which could occur due to previous coalescing.

The actual assignment first assigns physical registers their physical register and then allocates a physical register for each virtual register in turn, based on the order imposed by the stack:

```
bool assign_registers(RegAllocState *ras) {
   for (size_t i = 0; i < ras->first_vreg; i++) {
      ras->reg_assignment[i] = i;
   }

   bool have_spill = false;
   Oper u;
   while (wl_take_back(&ras->stack, &u)) {
      Oper reg = find_register_for(ras, u);
      if (reg == R_NONE) {
         spill_virtual_register(ras, u);
         have_spill = true;
         continue;
      }
      ras->reg_assignment[u] = reg;
   }

   return !have_spill;
}

void spill_virtual_register(RegAllocState *ras, Oper u) {
   ras->to_spill[u] = mfunction_reserve_stack_space(ras->mfunction, 8, 8);
}
```

Actual spills are marked by having a stack slot allocated.

At this point the assignment of virtual registers to physical registers exists only as a mapping in `ras->reg_assignment`. The mapping is applied by a separate function, which iterates over all instructions in a function and applies the mapping, i.e. for each register operand it does something like this:

```
inst->ops[i] = ras->reg_assignment[get_alias(ras, inst->ops[i])];
```

Again, resolution of aliases is important, since the nodes that got combined into another nodes are not allocated registers at all (never pushed onto the stack) and don't have valid entries in `reg_assignment`.

4.9.5 Interference graph representation

Graphs can be represented in different ways. Different representations are suitable for different purposes. In our case, we have seen how we use the interference graph in previous subsections:

1) Checks whether two nodes are neighbours in `are_interfering`.
2) Iteration over all neighbours. Needed for register assignment (finding out free registers amongst neighbours) and coalescing (for adding neighbours of one node to another).

Common representations like adjacency matrix support 1) efficiently, but not 2). On the other hand adjacency lists support 2) efficiently, but not 1). For this reason, usually *both* representations were historically used for interference graphs [Chaitin, 1982], and we do the same in our implementation.

The adjacency list representation has for each virtual register an array with all adjacent (neighbouring) virtual registers. In our implementation, these arrays are dynamic (i.e. the array can be reallocated and has a capacity as well as current size), because we extend the adjacency lists while coalescing.

An adjacency matrix can be efficiently represented as a *bit matrix* where the adjacency of two nodes is represented with a single bit. With a help of a mapping function, the matrix can actually be stored as a bit array and in our case can be smaller than $n \cdot n$ (where n is the number of nodes), since the adjacency matrix of an interference graph is symmetric and there aren't be self edges.

Other representations have been used as alternatives to adjacency matrix, for example hash tables [George et al., 1996] or "split bit-matrix" [Cooper et al., 1998], which trade lookup speed for memory. Memory consumption is a big disadvantage of the bit matrix, since its size is still $O(n^2)$ and in practice, it is very sparse. So far, we stayed with the bit matrix. In case need for reducing memory consumption arises, we can try other tricks, like compacting the virtual registers (reindexing them)—many temporaries are optimized out before register allocation, and by compacting we can decrease n and thus the size of the matrix considerably.

Our implementation maps the bit matrix onto a generic *bit set* (bit array):

```
bool are_interfering(RegAllocState *ras, Oper u, Oper v) {
   if (u == R_NONE || v == R_NONE)
      return true;
   Oper index = bitmatrix_index(ras, u, v);
   return bs_test(&ras->matrix, index);
}
```

Inspired by [George et al., 1996] we don't keep the adjacency lists for physical registers. The reasons for this are the following:

- Adjacency lists are needed in the register assignment, where we find free physical registers for virtual registers. This is not needed for physical registers.
- Adjacency lists for physical registers grow fairly big as physical registers are used across the whole function. For example the register `rbp` (used as a frame base pointer on the x86-64) practically interferes with *all* other registers.

Thus we special case physical registers when adding interference:

```
void add_interference(RegAllocState *ras, Oper u, Oper v) {
   if (u == v || are_interfering(ras, u, v))
      return;
   Oper index = bitmatrix_index(ras, u, v);
   bs_set(&ras->matrix, index);
   if (!is_physical(ras, u))
      garena_push_value(&ras->adj_list[u], Oper, v);
   if (!is_physical(ras, v))
      garena_push_value(&ras->adj_list[v], Oper, u);
   ras->degree[u]++;
   ras->degree[v]++;
}
```

4.9.6 Physical registers, coalescing

For register assignment (section 4.9.4) and interference graphs (section 4.9.5) we did introduce special cases for physical registers. Surprisingly enough most of the simplification and coalescing implementation details (section 4.9.3) didn't so far have *any* check for physical registers. Even though intuitively, physical registers are very much a special case:

- We shouldn't ever coalesce two physical registers or spill a physical register.
- When combining physical and virtual register, they should be combined into the physical register, because it has special meaning, for calling conventions etc.
- We shouldn't "simplify" and push physical registers onto the stack, since we want to only allocate virtual registers.
- Since we don't store adjacency lists for physical registers, we should never try to access them.

In practice, most of the issues are solved implicitly by adding interferences among all physical registers:

- Because physical registers interfere with each other, they are not allowed to coalesce together.
- If the number of physical registers is p, then each physical register has hast at least $p - 1$ interferences. This means that after all simplifications of virtual registers, physical registers would be simplified. In practice the number of available registers (k) is strictly less than the number of physical registers. For example, on x86-64 we reserve `rbp` and `rsp` for special purposes and don't allocate them. Even on other architectures usually at least one register is reserved for special purposes.

Other issues require more handling and care. For example, to always combine physical and virtual register into the physical register, we always combine into the register that compares smaller (physical registers are represented by smaller integers than virtual registers). Also when initializing the work lists, we don't put physical registers to the spill work list, which means they will never even be considered for spilling.

Another issue is the coalescing heuristic. Briggs' conservative coalescing heuristic requires iteration over adjacent nodes. Because we don't store adjacency lists for physical registers we have to use a different heuristic. George's heuristic is given (and proved safe) by [Appel et al., 1998]: "nodes a and b can be coalesced if, for every neighbor t of a, either t already interferes with b or t is of insignificant degree". The original [George et al., 1996] paper suggested the heuristic only for physical registers, but the heuristic can even be used even for virtual registers. In fact trying both heuristics (when possible) can be beneficial, since both are conservative and can detect safe coalescing in different cases.

We try both heuristics for virtual registers. In the implementation we assume that if one of the two registers is physical, it is `u`, which has to be ensured by the caller:

```
bool are_coalesceble(RegAllocState *ras, Oper u, Oper v) {
   bool coalescable = george_heuristic(ras, u, v);
   if (!coalescable && !is_physical(ras, u))
      coalescable = briggs_heuristic(ras, u, v);;
   return coalescable;
}
```

4.9.7 Spilling

When we need to select one out of multiple virtual registers as a potential spill (which might become actual spill later), then we want to spill the register, which is the *best* candidate for spilling. In our implementation the function `choose_and_spill_one` chooses the best candidate, and marks it as potential spill. In this section we discuss how the spill decision is made.

When choosing a spill we want to balance two effects of spilling a node:

1. *Negative effect.* When a node is spilled, we have to introduce a load for each its use and a store for each of its definitions. Statically, each load and store is equivalent, since they are just instructions, but dynamically, at runtime, the loads and stores may execute different number of times. In particular, memory operations inside of loops are likely to get executed more times compared to loads and stores in prologue and epilogue (such as those of the callee saved registers, see section 4.9.2.1). To minimalize the negative effect, we want to spill nodes with the *lowest* number of dynamic uses and definitions.
2. *Positive effect.* In a graph coloring register allocator, we model a spill of a node through its removal from the graph (see section 3.8.4.3 for the justification). But the important aspect from the point of the simplification is, that this reduces the degree of all of its neighbours by one, leading to further simplifications. To exploit the positive effect, we want to spill nodes with the *highest* degree.

We will call the negative effect *spill cost*, because it reflects the actual cost of the spill imposed on the program and we want to minimize it. We call *spill metric* the final metric in which we balance the *spill cost* (negative effect) and *degree* (positive effect).

In a graph coloring register allocator, the interference graph doesn't capture control flow at all, it is control flow insensitive. By considering control flow in spill choice we gain at least some control flow sensitivity.

4.9.7.1 Calculating spill cost

We have to calculate the cost of the inserted loads and stores statically, as we don't know how the code will behave dynamically at run time. However, we can be sure that instructions in the same basic block have the same cost, since they are always executed all or none. Additionally, instructions in loops should be more costly.

[Briggs et al., 1994] weights instruction by 10^d, where d is the *loop nesting depth* of the block which contains the instruction. This means that costs of instructions outside of loops are 1, inside a loop 10 and inside a doubly nested loop 100 etc. This method essentially assumes that each loop is executed 10 times.

With weights of instructions, we need to calculate the actual costs. For each use and definition of a register we should accumulate the sum of costs of all loads and stores that would be needed. In case of copy instructions, we don't need to keep the copy instruction after either load or store is inserted (see section 3.8.2.2), so spills in copy instructions are somewhat cheaper—while a memory operation is added, a copy instruction will be deleted.

For debugging purposes we accumulate costs of stores, loads and moves separately:

```
⟨for each instruction inst in machine block mblock⟩ {
   u16 cost = cost_in_depth(block->depth);
   ⟨for each definition def in inst⟩
      ras->def_cost[def] += cost;
   ⟨for each use use in inst⟩
      ras->use_cost[use] += cost;
   if (is_move(inst)) {
      ras->move_cost[inst->ops[0]] += cost;
      ras->move_cost[inst->ops[1]] += cost;
   }

}
```

Also, since we store the depth d, not 10^d, it is more convenient for use to actually use 8^d [Muchnick, 1997], which can be calculated more efficiently using shifts:

```
u32 cost_in_depth(u32 depth) {
   return 1 << (3 * depth);
}
```

4.9.7.2 Unspillable nodes

In section 3.8.2.1 we identified that some virtual registers are never worth spilling. Often these are virtual registers introduced by spill code. But also other registers which have a very close definition and use where spilling doesn't decrease register pressure. As they can have lower spill costs than other registers (because of low number of uses and definitions) we want to mark these nodes unspillable, to make sure we don't spill them.

Unspillable nodes have been identified already by Chaitin [Chaitin, 1982]. However, in our implementation we use formulation based on Briggs' [Briggs, 1992], which translates well to an actual implementation:

Definition 4.2. If some virtual register v becomes dead in between a use and associated definition of register u, then we call u's live range *interrupted.*

Definition 4.3. If none of the live ranges of virtual register u are interrupted, we call the virtual register *uninterrupted.*

We want to detect uninterrupted virtual registers and mark them unspillable. As the interruptions are based on liveness, we want to detect the property in a backwards pass over basic blocks. Apart from live-set the algorithm also maintains set of registers which are *currently uninterrupted* and the set of registers which have *ever been interrupted*, then for each instruction:

- If we reach a definition of a register, its live range ends and it is removed from the live-set. If additionally it is in the uninterrupted set, and not set as ever interrupted, then it is marked unspillable.
- Before the uses in the current instruction are added to the live-set, we check whether they are already in the set. If they are not live, this means that we have found a *death* of register—an interruption. We should mark all registers in the uninterrupted set as interrupted at least once, and reset the set.
- Only now we add all uses to the live-set and the currently-uninterrupted-set.

This solves local liveness. For registers live across basic block boundaries, we currently don't try to mark them uninterrupted, because expensive data-flow analysis would have to be performed. Instead, we are conservative and mark all registers initially in live-out as interrupted. This can mean that we may accidentally spill them, even though it wouldn't help, but in practice in the implementation we try to keep basic blocks maximal (i.e. if one block has one successor and the successor has only one predecessor we merge them), and the interesting unspillable situations occur with physically adjacent instructions, which make local analysis sufficient enough.

4.9.7.3 Spill metric

Spill costs are mostly static from the perspective of the simplification process—they are determined by the uses and definitions, which are not changed, except for the possible coalescing, which should combine the costs of the two nodes.[1] But the *degrees* of nodes are more dynamic, and are changed with every simplification.

Because of this, we combine the static costs with the degrees only when a spill candidate has to be selected. The actual calculation of the *spill metric* is thus following:

```
double spill_metric(RegAllocState *ras, Oper u) {
   if (wl_has(&ras->unspillable, u)) {
      return 1 / 0.0;
   }
   Oper cost = 2 * ras->def_cost[u] + 2 * ras->use_cost[u] - ras->move_cost[u];
   return (double) cost / ras->degree[u];
}
```

We weight the possible future stores and loads two times as much as the copy instructions, because they are more expensive. The metric is the quotient of the cost and the degree and in `choose_and_spill_one` we spill the virtual register with the lowest metric:

```
Oper candidate;
double max = 1 / 0.0;
⟨for each element u in work list ras->spill_wl⟩ {
   double curr = spill_metric(ras, u);
   if (curr < max || (curr == max && u < candidate)) {
      max = curr;
```

[1] Currently in the function `combine` we add the spill costs together, although this is imprecise, since we do not account for the move instructions that can be deleted due to the coalescing. More precise tracking of especially the move costs could also help with choice of freeze candidates, but would make the simplify loop more complex by including additional bookkeeping depending on both the instructions and their nesting depth in the control flow graph.

```
        candidate = u;
    }
}
assert(max < 1 / 0.0);
```

The inverse of the spill metric, where we would divide by the cost, instead of by the degree, is not as practical—depending on our approach to definitions and uses the cost could actually even get negative or zero (not in our implementation though, where defs and uses incorporate the moves, and the moves just reduce the cost). Unfortunately, the minimum formulation makes it slightly harder to pick the candidate with lowest metric, as the metric of the first node in the work list cannot be used as a reference point, because it may actually be *unspillable*. For this reason we use the `inf` double value created by division by zero.

In an extreme case, *all* nodes in the work list may be marked unspillable, we assert that this does not happen and that we have at least one spillable candidate that we can actually spill.

Spilling an unspillable candidate is wrong, since it will just be replaced with a fresh temporary that also is unspillable and we might easily get into an infinite loop of unproductive spills. The fact that spill of unspillable won't be attempted should be guaranteed statically. In practice, the assertion can find issues where the back end programmer mistakenly makes too few registers available (i.e. we can't expect the register allocation to succeed with 1 available register, when most operations require two operands, assuming we don't change the instructions while spilling; see section 3.8.2.1), or makes the registers unavailable with a mistake in an implementation of calling conventions (see section 4.9.2.1).

As a special case, if two virtual registers have the same metric, we choose the one which is a smaller number. This is mostly so we can primarily spill the register used to hold the `rbx` callee saved register as opposed to for example `r12` (another callee saved register), because use of `rbx` can sometimes save a REX prefix byte (see section 2.4).

4.10 Runtime

Programs output by our compiler like all non-toy programs will need to interact with the operating system in two crucial ways:

- The operating system needs to transfer the control to the entry point of the program.
- The program will need to call routines provided by the operating system.

Even successfully exitting the program involves a request to the operating system (e.g. through a call to the `exit` function of standard C). As mentioned in section 2.6, on most platforms the C library is the *only* way to make requests to the operating system. On others, it is possible to perform syscalls directly if they have stable system call interface. Linux is known for such stability, but most applications still perform system calls through a C library, such as glibc[1].

Startup is also usually handled by the C library. The program loader in the Linux kernel jumps to the entry point address listed in the ELF binary, and while it passes some data to the program by writing it to its address space, it doesn't call `main` nor prepares its arguments. The transition from entry point to `main` is handled by the *runtime component*[2], which is linked into the application.

[1] `https://www.gnu.org/software/libc/`

[2] In C it is sometimes associated with the acronym CRT—*C runtime*.

In following subsection, we demonstrate how TinyC programs are able to be linked with the system C library, and call its functions and how TinyC programs can perform system calls directly.

Our implementation supports both behaviours. Though performing system calls directly is a bit bare bones, as no further abstractions are provided at the moment. Both of the approaches are mainly intended to showcase how TinyC the language would need to be extended, since as of now it doesn't support calls to external functions or variable number of arguments.

4.10.1 External C library

As we fully implement the System V ABI. Programs produced by us are fully able to use the existing C libraries. They still have to be linked correctly to the C library's runtime. We shall consider the following example:

```
// prog.c
int puts(char *s);

int main(int argc, char **argv) {
   puts(argv[0]);
   return 0;
}
```

There are several things to note:

- We declare the **puts** function to make our TinyC compiler aware of this external function. As it won't see any definition of it, it will presume it to be an external function.
- We define a **main** function receiving two arguments. First one is 32-bit, passed in register **rdi**, second is 64-bit pointer passed in **rsi** and the return value is to be saved into **rax**.
- The external function **puts** gets called with the 0th argument (the program name).

Translated to NASM assembly (with unrelated output omitted):

```
; prog.asm
   extern puts
   global main
main:
.L0:
   push rbp
   mov rbp, rsp
   mov rdi, [rsi]
   call puts wrt ..plt
   xor rax, rax
   mov rsp, rbp
   pop rbp
   ret
```

- The external puts function is marked as *external*, for similar reasons it was in TinyC: the function will be provided externally and only resolved when (dynamically) linked in.
- The **main** function is marked as *exported*. This makes the label into a symbol in the resulting object file, and thus it will be findable by other linked object files or libraries, like the C runtime, which calls it.

- The call to `puts` is marked with `wrt ..plt` which makes NASM emit the call as call through the *procedure linkage table*. We do this to get a *position independent executable*.

The assembly can then be assembled into an ELF object file with `nasm`:

```
nasm -f elf64 -o prog.o prog.asm
```

The final executable is then linked together with the system linker `ld`:

```
ld -pie -m elf_x86_64 -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o prog
   /usr/lib/Scrt1.o
   prog.o
   /usr/lib/libc.so
```

Apart from requesting linking into a position independent executable ELF x86-64 executable, the ELF interpreter ("dynamic linker") is set to its standard path [System V AMD64 ABI, 2023]. The following three ELF files are linked together:

- `Scrt1.o`, which contains the `_start` symbol. `_start` is marked as entry point by the linker's default linker script. The final address of `_start` will be marked in the ELF as the entry point, i.e. the address where the kernel will jump to.
- `prog.o`, our object file, which contains `main`.
- `/usr/lib/libc.so`, the dynamic library (*shared object*) version of the C standard library, containing `puts` amongst others. Will be only linked at runtime by the dynamic linker.

The program can then be run like this:

```
./prog
```

In reality, there are usually more object files linked in. Notably they handle the runs of constructs and destructors (code which runs before and after main, respectively). Since our program didn't need them, we didn't need to link them. But the C dynamic library *could* need them. We are in dangerous zone of compiler and C library specific details. It is usually better to use the C compiler's driver to figure out the linker command line. Such driver is the `gcc` command of the GCC compiler. With the driver, it is also much easier to use *static linking* or an alternative library like *musl*:

```
gcc -o prog obj.o
gcc -static -o prog obj.o
musl-gcc -o prog obj.o
musl-gcc -dynamic -o prog obj.o
```

The actual linker invocation can be then found by adding the `-v` flag.

4.10.2 Custom runtime

An equivalent program can be created without using the C library. For simplicity and presentation purposes, we will substitute the `puts` function with a simple write of 6 characters (without a new line), to avoid having to figure out the length of the program name, etc. We will just illustrate the runtime portion which will run the program's `main` function, perform a system call and exit the program with `main`'s return value as exit code.

The most important thing is the entry point. We can define this with the name `_start` as the linker defaults to it, but we could choose any other name, as long as we export the symbol and instruct the linker correctly.

According to the System V ABI [System V AMD64 ABI, 2023], the entry point finds itself with the stack aligned to 16 bytes, with the 8-byte argument count on top of the stack (`[rsp]`) and the pointers to arguments start after it (`[rsp+8]`). We can prepare these to appropriate registers to be passed to `main`. Then after `main` is called, we need to run the `exit` system call, which has the number 60 (to be passed in `rax`) and the argument (`rdi`) is the exit code, which `main` returns in `rax`. Additionally, the ABI says that `rbp` is unspecified at entry and should be zeroed out to mark the outermost stack frame and we abide:

```
   global _start
_start:
   xor rbp, rbp
   mov rdi, [rsp]
   lea rsi, [rsp+8]
   call main
   mov rdi, rax
   mov rax, 60
   syscall
```

Since we are guaranteed that the stack is aligned to 16 bytes, we don't have to realign it before calling `main` (which, like all functions respecting the System V ABI, expects the stack to be 16 byte aligned).

To add system call support, we need to translate from the System V *C ABI* to the System V *Linux kernel ABI* and to perform the system call with the `syscall` instruction, which is not normally accessible to C programs. We do this with a syscall *function* hand-coded in assembly. The function will receive up to 7 arguments, the first one will be the system call number (to be put into `rax`) and then the 6 arguments, which the kernel expects in registers. As our `syscall` function will be called from C code, the first 6 arguments are passed in registers and the 7th on the stack—we are off by one, since we also receive the system call number. We also fix the difference in the two calling conventions, where the kernel uses `r10` instead of `rcx`:

```
syscall:
   mov rax, rdi
   mov rdi, rsi
   mov rsi, rdx
   mov rdx, rcx
   mov r10, r8
   mov r8, r9
   mov r9, [rsp+8]
   syscall
   ret
```

The simple definition actually packs a few other things that are not immediately apparent:

- The function works for any number of arguments. Six is the maximum number of arguments any system call has. If a syscall needs less than 6 arguments, the extraneous parameters simply won't be read. This is also the reason why the read of the 7th parameter from the stack is benign. The argument registers don't have to be preserved according to the C calling convention, so it doesn't matter that we change them.
- Callee saved registers (`rbp`, `rbx`, and `r12` through `r15`) are preserved, since neither our function nor the kernel changes them. (The kernel changes only `rcx` and `r11` and only due to them being used implicitly by the `syscall` instruction, see section 2.6).

- The return value from the kernel (`rax`) is passed back in the C return register (`rax`).

Essentially we have a variable argument function without using `va_list`. But the caller can still pass us arguments using the C ABI and as long as there are no more than 7, we can understand them (and there shouldn't actually be more than 7). In TinyC we can make the function available with the following declaration:

```
int syscall(int syscall_nr, ...);
```

To actually write something to standard output, we can for brevity at least output the first 6 characters (`./prog`) of the program name:

```
int main(int argc, char **argv) {
   syscall(1, 0, argv[0], 6);
   return 0;
}
```

Here 1 corresponds the `write` system call and 0 to the file descriptor of standard output. A wrapper for the `write` system call, can be created like this:

```
int write(int fd, void *buf, int count) {
    return syscall(1, fd, buf, count);
}
```

With wrappers for all usual system calls, we can hide the ugly implementation details from the users.

A few wrappers for system calls like `open`, `close`, `read`, `write` (input and output) `mmap`, `munmap` (memory mapping, allocation) and `exit` go a long way towards a base on top of which a C library could be implemented.

4.11 Middle end

As no complete front end nor middle end for TinyC was available when work on this thesis was started, we created our own.

The front end is a simple recursive descent Pratt parser (see [Pratt, 1973]). In a single pass it parsers, type checks and emits middle end IR (constants and operations as described in section 4.3.1).

More interesting things happen in the middle end, where we do a few optimizations that our back end isn't able to currently do.

4.11.1 Jump threading

We perform a bit of *jump threading* optimization, which removes some control flow edges needlessly inserted by our simplistic front end. Sometimes by the threading we we introduce *critical edges*. We split them again later (see section 4.4), but keeping them unsplit at least temporarily allows for simpler control flow on which we do the next optimization.

4.11.2 Value numbering SSA construction

We perform a very weak form of "alias analysis" on stack slots (what used to be TinyC local variables) and in simple cases are able to optimize them into values respecting SSA, replacing the former loads and stores.

Since values in our implementation translate to virtual registers, this essentially means that we replace memory operations by operations on registers. Although this increases register pressure, the introduced spills are not worse than the previous memory operations.

The simple alias analysis notices that any stack slot, which is only involved in operations that either load or store to it, can be treated as a simple variable on which we can apply SSA construction. Other uses of the stack slot *could* mean that the stack slot value (which is a constant, pointer to the stack slot), is *aliased* and written to by other stores. This analysis is very weak, as it aborts in cases where there are in fact no aliases. But in our experience it is powerful enough and is able to eliminate many *scalar* "stack allocated variables". Our analysis doesn't attempt to optimize *aggregates* (structs) or *arrays*.

When optimizable stack slots are identified, the stores and loads are regarded as assignments and reads, which can be used with any SSA construction method. We use Braun's method [Braun et al., 2013], which can be used for SSA repairs. The algorithm is conceptually simple and intuitive: it works like global value numbering, but reads ϕ-function operands by recursively querying predecessor blocks. Handling of cycles and speed are achieved with elegant memoization.

Chapter 5
Evaluation

Our back end does global register allocation and extensive, but mostly local peephole optimization. There are no big transformations of control flow or analysis, that could achieve *optimization*.

Our assumption is, that the code is already fairly optimized when provided as input to the back end. Our back end then does its best to generate the best sequence of machine instructions implementing the exact same behavior, but with x86-64 specific improvements, like use of the more complex and compact addressing modes or improved work with flags.

There are optimizations that might be better suitable for a back end.

5.1 Known shortcomings

Things that are known not to produce good code include:

- Division by a (power of two) constant. Divisions by constants, are much better realized with multiplication by reciprocal or, in case of powers of two, by shifts. Unfortunately, due to register constraints of the `idiv` instruction, detecting divisions by constants is not as easy as it may seem.
- Identities on comparisons. Simple identities like `a != 0` are optimizable quite well in our back end, and can be also fused into conditional jump instructions. But any more complex comparison expressions like `(a != 0) == 1` are not easily folded by our peephole optimizer. This is because such operations compile into quite a few instructions (as the `setcc` instruction sets only a byte, which has partial dependency problems), and reads and writes of flags are not as easy to optimize in our current design.
- Extensive use of narrow integers. We support TinyC `char`s purely with sign extending loads and trimming stores. This is in line with normal C semantics, which promote narrow types to `int` before doing any operations. However, sometimes it is possible to operate on memory directly e.g. with an instruction like: `add byte [rax], 1`. As we only support narrow types for loads and stores, we are not able to do this optimization. It would be possible with more general handling of narrow operations, which does not tie them to subkinds.

We hope that the constants propagation, constant folding and arithmetic/comparison identities get applied in the middle end, which is also much more suited to actually find the constants to optimize with, such as with the sparse conditional constant propagation optimization [Wegman et al., 1991].

There are surely many other shortcomings, but these are the ones we encountered a few times and know are the limitations of our architecture, and can't be fixed by adding a simple two-instruction peephole pattern.

5.2 Comparison against GCC

Since our source language is based on C, and we output x86-64 machine code, we are able to compare our code with the output of existing C compilers, like GCC[1].

Comparison with GCC is especially interesting, since it's back end representation (RTL, the *Register Transfer Language*) is based on the Davidson-Fraser (see section 3.6.2), which inspired our successive peephole optimization of machine code. Though our implementation doesn't use actual low level register transfers and has linear sequences of instruction, while GCC's RTL is actually tree based.

Unfortunately, it is not straightforward to compare just the back ends. As both compilers (ours and GCC) have very low level and different program representations, we can't even generate a code that would be an equivalent starting point for both. Our compiler itself has only few middle end optimizations (see section 4.11).

Still we decided to do at least some comparison against GCC. We focused on a simple implementation of the Sieve of Eratosthenes. It exercises quite a few components of the implementation:

- `char`s and bit arrays from `int`s,
- Plenty constants and memory addressing for better use of addressing modes.
- Control flow in doubly nested loops. If value numbering SSA construction (section 4.11.2) is used, then placement of cyclic ϕ-operations in the doubly nested loops is tested.

As a benchmark, the Sieve is mainly CPU bound, not I/O bound, which is a must for any similar benchmark. We implemented three versions of the sieve in two languages—TinyC and C. The benchmarks had to be implemented separately, to keep the behavior the same. For example, while TinyC integers are 64-bit, the equivalent type in C on x86-64 is `long`. The three different versions were:

- `sieve`. Using `char` array with a bit per byte.
- `sieve-bit`. Using `int` (64-bit integer) array with a 64-bits per element.
- `sieve-bit-cse`. Using `int` (64-bit integer) array with a 64-bits per element. Additionally, eliminates a common subexpression.

The first version is the real benchmark we started with. Then we investigated how the two compilers would fare with a fairly high number of bit twiddling operations. In the bit set implementation, there was also a common subexpression, that our compiler is not able to eliminate. We eliminated it manually, to try to aid comparison. The results are in table 5.1.

Table 5.1. Benchmark of Sieve of Eratosthenes [ms]

	GCC			Our implementation			
	`-O2`	`-O1`	`-O0`	P, V	V[i]	P[ii]	–
sieve	1477	1475	2114	1622	1751	2066	2730
sieve-bit	954	931	1915	1475	2009	1905	3060
sieve-bit-cse	951	935	1918	1269	1760	1938	2705

[i]Value numbering SSA construction enabled
[ii]Peephole optimization enabled

[1] `https://gcc.gnu.org/`

We tested with various optimization levels in both GCC and in our implementation. Since our implementation only has two important optimizations (value numbering SSA construction in the middle end and peephole optimization in back end), we tested all combinations.

We can see, that the unoptimized version of our compiler was still quite a bit slower (2730 ms) than unoptimized GCC version. After looking at the assembly, it makes sense. Without optimization of stack slots, both GCC and our implementation do a lot of memory operations. But our very naive code generations creates a lot of very verbose code, that just runs slower. We haven't really intended for peephole optimization to be omitted.

With stack slot elimination, and peephole optimization, we get almost within 10 % of GCC, which seems to not be able to make more optimizations to the code after the `-O1` level. Surprisingly, the `-O1` is consistently a bit faster, but the since the observed deviation is about 8 ms, we can also attribute it to measuring error. No immediately obvious differences are in the `-O2` and `-O1` code. Looking at the assembly of our compiler's optimized version, it looks like a version somebody would write by porting the C code to assembly—there are no excessive uses of stack, the register use looks reasonable, and the control flow is one-to-one the same as in the TinyC version, which makes sense, since we don't do any big control flow transformations.

The big amount of eliminated memory operations is likely the reason why SSA construction fares much better individually, then peephole optimization for the `sieve` benchmark. Even with our peephole optimizer we don't get much faster then GCC's `-O0`. It seems that with GCC's `-O0` doesn't do any stack slot elimination, but there are still apparent uses of advanced addressing modes e.g. `lea` is used for multiplication by 8.

For the second round of benchmarks with bit set implemented with 64-bit integers, we see that GCC is able to be much faster then we are. GCC is much better at improving the control flow and the use of arithmetic instructions. But an improvement can be seen in the third benchmark, which has the common subexpression eliminated. This removes a fair amount of shifts, as the common subexpression looks like this (and is in a hoot loop):

```
not_prime[i >> 6] = not_prime[i >> 6] | (1 << (i & 63));
```

This is easily fixable in ordinary C, where there are arithmetic assignment operators. In TinyC we simulated the with the following:

```
int *prim = &not_prime[i >> 6];
*prim = *prim | (1 << (i & 63));
```

The elimination depends on elimination of the temporary from stack to actually be more efficient. Which is why we see that the peephole optimizing version is a bit slower on the CSE benchmark.

We think our back end did well. Most of the difference in speed seems to be coming from the middle end.

Chapter 6
Conclusion

In this thesis we studied x86-64, one of the most prevalent architectures today. We designed and implemented a back end for translating TinyC IR into x86-64 machine code. To achieve this, we investigated existing literature on the topic of compiler back ends. This final chapter provides some reflections, practical considerations and ideas for future work.

The x86-64 features many constraints that make it harder to compile for. A lot of these constraints can be overcome with the right use of live range splitting. The problems with register classes on x86-64 can be avoided by not using problematic registers (like `ah`)—this not only makes register allocation simpler, but the resulting code is also also faster in practice, because use of the historical register classes is associated with big penalties on modern superscalar processors.

Irregularities in the x86-64 instruction set and the addressing modes make it harder for a compiler to model instruction selection with trees or DAGs. In our design, we chose to first do a naive translation to x86-64 machine instructions and then optimize the generated code successively with *peephole optimizations*. This is an application of the Davidson-Fraser model [Davidson et al., 1984a], though we work on instructions directly, instead of on a separate low level intermediate representation. In this model we can easily express the constraints mentioned above—our naive code generation provides correctness, while the optimizations make the produced code efficient.

Peephole optimization itself proved to be capable of improving the code to take advantage of the various x86-64 addressing modes, though data-flow based optimizations have to be used, otherwise many optimizations can be missed due to not falling into a peephole window. Writing the peephole patterns by hand, is tractable for a single architecture, but it is error prone—patterns are not too hard to create in isolation, but their interactions can sometimes become problematic. Some automation or a domain specific language for writing the patterns would be a must for a compiler with more target architectures or a big number of patterns. On the other hand, hand-written patterns proved to be quite expressive and can easily cover more cases then automatically derived patterns could.

Modern optimizing compilers use SSA form in their middle ends. Deconstructing the ϕ-functions (not executable by machines) into copy instructions is not as trivial as it may seem, but correct and simple algorithms are available.

Live range splitting, compilation to x86-64's two address code as well as SSA deconstruction can all produce a large number of copy instructions. Coalescing, which is able to remove the copy instructions if possible, is needed to produce efficient code. Coalescing has to be done carefully to ensure both correctness as well as code quality, since excessive coalescing can lead to spilling. This is one of the reasons why we used the iterated register coalescing algorithm [George et al., 1996] for register allocation—it is based on the classic well known technique of graph coloring register allocation, but is able to eliminate many more copy instructions than previous approaches.

Although the produced code is good for code sequences that are covered by our peephole patterns, the code quality goes down significantly if our patterns are weak enough and not able to optimize a sequence of instructions. A lot of times this can be attributed to translation from middle end IR that was not sufficiently optimized—the inoptimalities are only magnified in our back end, which is mostly focused on making use of x86-64's features as much as possible. The middle end is much more suited for optimizations that are not x86-64 specific, not only because improvements there apply to other architectures, but also because it works with a higher level abstractions and still has some knowledge about semantics, while the backend already works mostly with untyped integers.

This thesis, both the text and the actual implementation, can hopefully serve as examples to the students at FIT CTU considering advancing their compilers beyond the simplified Tiny86 architecture.

The implementation of the back end, documentation, as well as the source of this text is available publicly under an open source license.[1].

6.1 Future work

The back end works, and can be used as is. However there are still some interesting areas that could be explored in future work.

We tried to design the back end with porting to other target architectures in mind. It would be great to exercise that in practice and add a second target architecture to our compiler. Starting without a peephole optimizer, the port should hopefully be relatively straightforward. Porting to an architecture like AArch64 would be the most interesting, since it features register classes that require more care than the ones on x86-64.

Ports to other calling conventions (like Microsoft's) or to operating systems which require system calls to be performed directly (like Windows or OpenBSD) would exercise other parts of the compiler.

The back end produces textual NASM assembly, that can be turned into a final executable with external programs (NASM assembler and system linker). This works great for inspection and allows linking with external code, but still shows all important aspects of the compiler. Making the back end produce executables directly could however make it more self-contained as external dependencies would no longer be required.

As of now, the back end is implemented as a standalone program. Turning it into a library could make it more universally usable.

We also identified a few extensions to the TinyC language (external calls, variable number of arguments) that are needed in order to run non-trivial programs on the x86-64 architecture.

With experience with both x86-64 and Tiny86, we see a few areas where Tiny86 could be made more realistic (system calls) or improved a bit (unsigned operations).

Extensions to both TinyC and Tiny86 are not hard to implement, but they have to be carefully evaluated before they needlessly complicate things for students.

[1] `https://github.com/vlasakm/master-thesis`

References

Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions.* 2022 [cit. 2023-06-09]. Available from `https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volume-3-general-purpose-and-system`.

Aho, Alfred V., and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery, jun, 1975, Vol. 18, No. 6, pp. 333–340. ISSN 0001-0782. Available from DOI 10.1145/360825.360855. Available from `https://doi.org/10.1145/360825.360855`.

Aho, Alfred V., Mahadevan Ganapathi, and Steven W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, oct, 1989, Vol. 11, No. 4, pp. 491–516. ISSN 0164-0925. Available from DOI 10.1145/69558.75700. Available from `https://doi.org/10.1145/69558.75700`.

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

Appel, Andrew W., and Lal George. Optimal Spilling for CISC Machines with Few Registers. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery, may, 2001, Vol. 36, No. 5, pp. 243–253. ISSN 0362-1340. Available from DOI 10.1145/381694.378854. Available from `https://doi.org/10.1145/381694.378854`.

Appel, Andrew W., and Maia Ginsburg. *Modern compiler implementation in C.* Cambridge, UK: Cambridge University Press, 1998. ISBN 052158390X.

Belady, Laszlo A.. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal.* IBM, 1966, Vol. 5, No. 2, pp. 78–101.

Blindell, Gabriel Hjort. *Instruction Selection: Principles, Methods, and Applications.* 1st ed. Springer Publishing Company, Incorporated, 2016. ISBN 3319340174.

Boissinot, Benoit, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In: *2009 International Symposium on Code Generation and Optimization.* 2009. pp. 114-125. Available from DOI 10.1109/CGO.2009.19.

Bouchez, Florent, Alain Darte, Christophe Guillon, and Fabrice Rastello. *Register Allocation: What Does the NP-Completeness Proof of Chaitin et al. Really Prove? Or Revisiting Register Allocation: Why and How.* Available from DOI 10.1007/978-3-540-72521-3_21. Available from `https://doi.org/10.1007/978-3-540-72521-3_21`.

Braun, Matthias, Sebastian Buchwald, Sebastian Hack, Roland Leissa, Christoph Mallon, and Andreas Zwinkau. Simple and Efficient Construction of Static Single Assignment Form. In: *Proceedings of the 22nd International Conference on Compiler Construction*. Berlin, Heidelberg: Springer-Verlag, 2013. pp. 102–122. CC'13. ISBN 9783642370502. Available from DOI 10.1007/978-3-642-37051-9_6. Available from `https://doi.org/10.1007/978-3-642-37051-9_6`.

Briggs, Preston. *Register allocation via graph coloring*. Houston, Texas, USA: Rice University, 1992. Master's Thesis.

Briggs, Preston, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Softw. Pract. Exper.* USA: John Wiley & Sons, Inc., jul, 1998, Vol. 28, No. 8, pp. 859–881. ISSN 0038-0644.

Briggs, Preston, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, may, 1994, Vol. 16, No. 3, pp. 428–455. ISSN 0164-0925. Available from DOI 10.1145/177492.177575. Available from `https://doi.org/10.1145/177492.177575`.

Briggs, Preston, and Linda Torczon. An Efficient Representation for Sparse Sets. *ACM Lett. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, mar, 1993, Vol. 2, No. 1–4, pp. 59–69. ISSN 1057-4514. Available from DOI 10.1145/176454.176484. Available from `https://doi.org/10.1145/176454.176484`.

Cattel, R. G. G.. *Formalization and Automatic Derivation of Code Generators*. Pittsburgh, Pennsylvania, USA: Carnegie Mellon University, 1978. Ph.D. Thesis.

Chaitin, G. J.. Register Allocation & Spilling via Graph Coloring. In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. New York, NY, USA: Association for Computing Machinery, 1982. pp. 98–105. SIGPLAN '82. ISBN 0897910745. Available from DOI 10.1145/800230.806984. Available from `https://doi.org/10.1145/800230.806984`.

Chaitin, Gregory J., Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*. Elsevier BV, jan, 1981, Vol. 6, No. 1, pp. 47–57. Available from DOI 10.1016/0096-0551(81)90048-5. Available from `https://doi.org/10.1016/0096-0551(81)90048-5`.

Cooper, Keith D., Timothy J. Harvey, and Ken Kennedy. An Empirical Study of Iterative Data-Flow Analysis. In: *Proceedings of the 15th International Conference on Computing*. USA: IEEE Computer Society, 2006. pp. 266–276. CIC '06. ISBN 0769527086. Available from DOI 10.1109/CIC.2006.22. Available from `https://doi.org/10.1109/CIC.2006.22`.

Cooper, Keith D., Timothy J. Harvey, and Linda Torczon. How to build an interference graph. *Software: Practice and Experience*. Wiley Online Library, 1998, Vol. 28, No. 4, pp. 425–444.

Cooper, Keith D., and Linda Torczon. *Engineering a Compiler*. 2004.

Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, oct, 1991, Vol. 13, No. 4, pp. 451–490. ISSN 0164-0925. Available from DOI 10.1145/115372.115320. Available from `https://doi.org/10.1145/115372.115320`.

DAVIDSON, Jack W., and Christopher W. FRASER. The Design and Application of a Retargetable Peephole Optimizer. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, apr, 1980, Vol. 2, No. 2, pp. 191–202. ISSN 0164-0925. Available from DOI 10.1145/357094.357098. Available from `https://doi.org/10.1145/357094.357098`.

DAVIDSON, Jack W., and Christopher W. FRASER. Code Selection through Object Code Optimization. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, oct, 1984, Vol. 6, No. 4, pp. 505–526. ISSN 0164-0925. Available from DOI 10.1145/1780.1783. Available from `https://doi.org/10.1145/1780.1783`.

DAVIDSON, Jack W., and Christopher W. FRASER. Automatic Generation of Peephole Optimizations. In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction.* New York, NY, USA: Association for Computing Machinery, 1984. pp. 111–116. SIGPLAN '84. ISBN 0897911393. Available from DOI 10.1145/502874.502885. Available from `https://doi.org/10.1145/502874.502885`.

ECKSTEIN, Erik, Oliver KÖNIG, and Bernhard SCHOLZ. Code Instruction Selection Based on SSA-Graphs. In: Andreas KRALL, ed. *Software and Compilers for Embedded Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. pp. 49–65. ISBN 978-3-540-39920-9.

ERTL, M. Anton. Optimal Code Selection in DAGs. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York, NY, USA: Association for Computing Machinery, 1999. pp. 242–249. POPL '99. ISBN 1581130953. Available from DOI 10.1145/292540.292562. Available from `https://doi.org/10.1145/292540.292562`.

FRASER, Christopher W., David R. HANSON, and Todd A. PROEBSTING. Engineering a Simple, Efficient Code-Generator Generator. *ACM Lett. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, sep, 1992, Vol. 1, No. 3, pp. 213–226. ISSN 1057-4514. Available from DOI 10.1145/151640.151642. Available from `https://doi.org/10.1145/151640.151642`.

FRASER, Christopher W., Robert R. HENRY, and Todd A. PROEBSTING. BURG: Fast Optimal Instruction Selection and Tree Parsing. *SIGPLAN Not.* New York, NY, USA: Association for Computing Machinery, apr, 1992, Vol. 27, No. 4, pp. 68–76. ISSN 0362-1340. Available from DOI 10.1145/131080.131089. Available from `https://doi.org/10.1145/131080.131089`.

GAMMA, Erich, Richard HELM, Ralph JOHNSON, and John VLISSIDES. *Design patterns: elements of reusable object-oriented software.* Westford, Massachusetts, United States: Addison-Wesley, 1995. ISBN 0-201-63361-2.

GEORGE, Lal, and Andrew W. APPEL. Iterated Register Coalescing. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, may, 1996, Vol. 18, No. 3, pp. 300–324. ISSN 0164-0925. Available from DOI 10.1145/229542.229546. Available from `https://doi.org/10.1145/229542.229546`.

GLANVILLE, R. Steven, and Susan L. GRAHAM. A New Method for Compiler Code Generation. In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on*

Principles of Programming Languages. New York, NY, USA: Association for Computing Machinery, 1978. pp. 231–254. POPL '78. ISBN 9781450373487. Available from DOI 10.1145/512760.512785. Available from `https://doi.org/10.1145/512760.512785`.

HACK, Sebastian, Daniel GRUND, and Gerhard GOOS. Register Allocation for Programs in SSA-Form. In: *Proceedings of the 15th International Conference on Compiler Construction*. Berlin, Heidelberg: Springer-Verlag, 2006. pp. 247–262. CC'06. ISBN 354033050X. Available from DOI 10.1007/11688839_20. Available from `https://doi.org/10.1007/11688839_20`.

INTEL CORPORATION. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2023a [cit. 2023-06-09]. Available from `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

INTEL CORPORATION. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 2023b [cit. 2023-06-09]. Available from `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

KAM, John B., and Jeffrey D. ULLMAN. Monotone Data Flow Analysis Frameworks. *Acta Inf.* Berlin, Heidelberg: Springer-Verlag, sep, 1977, Vol. 7, No. 3, pp. 305–317. ISSN 0001-5903. Available from DOI 10.1007/BF00290339. Available from `https://doi.org/10.1007/BF00290339`.

KNUTH, Donald E. Semantics of context-free languages. *Mathematical systems theory*. Springer, 1968, Vol. 2, No. 2, pp. 127–145.

KNUTH, Donald E.. On the translation of languages from left to right. *Information and Control*. 1965, Vol. 8, No. 6, pp. 607-639. ISSN 0019-9958. Available from DOI https://doi.org/10.1016/S0019-9958(65)90426-2. Available from `https://www.sciencedirect.com/science/article/pii/S0019995865904262`.

KOES, David Ryan, and Seth Copen GOLDSTEIN. Near-Optimal Instruction Selection on Dags. In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. New York, NY, USA: Association for Computing Machinery, 2008. pp. 45–54. CGO '08. ISBN 9781595939784. Available from DOI 10.1145/1356058.1356065. Available from `https://doi.org/10.1145/1356058.1356065`.

LEUNG, Allen, and Lal GEORGE. *A new MLRISC register allocator*. 1998.

LU, H.J., Michael MATZ, Milind GIRKAR, Jan HUBIČKA, Andreas JAEGER, and Mark MITCHELL. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 2023 [cit. 2023-06-12]. Available from `https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build`.

MICROSOFT CORPORATION. *x64 calling convention*. 2022 [cit. 2023-06-12]. Available from `https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170`.

MUCHNICK, Steven S.. *Advanced Compiler Design and Implementation*. San Francisco, USA: Morgan Kaufmann Publishers, 1997. ISBN 1-55860-320-4.

MÖSSENBÖCK, Hanspeter, and Michael PFEIFFER. Linear Scan Register Allocation in the Context of SSA Form and Register Constraints. In: R. Nigel HORSPOOL, ed. *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. pp. 229–246. ISBN 978-3-540-45937-8.

PALL, Mike. *LuaJIT 2.0 intellectual property disclosure and research opportunities* [online]. 2009 [cit. 2023-05-29]. Available from `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`.

PARK, Jinpyo, and Soo-Mook MOON. Optimistic Register Coalescing. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, jul, 2004, Vol. 26, No. 4, pp. 735–765. ISSN 0164-0925. Available from DOI 10.1145/1011508.1011512. Available from `https://doi.org/10.1145/1011508.1011512`.

PELEGRÍ-LLOPART, E., and S. L. GRAHAM. Optimal Code Generation for Expression Trees: An Application BURS Theory. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1988. pp. 294–308. POPL '88. ISBN 0897912527. Available from DOI 10.1145/73560.73586. Available from `https://doi.org/10.1145/73560.73586`.

PEREIRA, Fernando Magno Quintão, and Jens PALSBERG. Register Allocation Via Coloring of Chordal Graphs. In: Kwangkeun YI, ed. *Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. pp. 315–329. ISBN 978-3-540-32247-4.

POLETTO, Massimiliano, and Vivek SARKAR. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, sep, 1999, Vol. 21, No. 5, pp. 895–913. ISSN 0164-0925. Available from DOI 10.1145/330249.330250. Available from `https://doi.org/10.1145/330249.330250`.

PRATT, Vaughan R.. Top down Operator Precedence. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1973. pp. 41–51. POPL '73. ISBN 9781450373494. Available from DOI 10.1145/512927.512931. Available from `https://doi.org/10.1145/512927.512931`.

ROSEN, B. K., M. N. WEGMAN, and F. K. ZADECK. Global Value Numbers and Redundant Computations. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1988. pp. 12–27. POPL '88. ISBN 0897912527. Available from DOI 10.1145/73560.73562. Available from `https://doi.org/10.1145/73560.73562`.

SCHOLZ, Bernhard, and Erik ECKSTEIN. Register Allocation for Irregular Architectures. In: *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*. New York, NY, USA: Association for Computing Machinery, 2002. pp. 139–148. LCTES/SCOPES '02. ISBN 1581135270. Available from DOI 10.1145/513829.513854. Available from `https://doi.org/10.1145/513829.513854`.

SMITH, Michael D., Norman RAMSEY, and Glenn HOLLOWAY. A Generalized Algorithm for Graph-Coloring Register Allocation. In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2004. pp. 277–288. PLDI '04. ISBN 1581138075. Available from DOI 10.1145/996841.996875. Available from `https://doi.org/10.1145/996841.996875`.

SREEDHAR, Vugranam C., Roy Dz-Ching JU, David M. GILLIES, and Vatsa SANTHANAM. Translating Out of Static Single Assignment Form. In: Agostino CORTESI, and Gilberto FILÉ, eds. *Static Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. pp. 194–210. ISBN 978-3-540-48294-9.

STREJC, Ivo. *Tiny x86 - Architecture Simulator for Educational Purposes*. Prague, Czech Republic: Faculty of Information Technology, CTU in Prague, 2021. Master's Thesis. Available from `http://hdl.handle.net/10467/94644`.

TRAUB, Omri, Glenn HOLLOWAY, and Michael D. SMITH. Quality and Speed in Linear-Scan Register Allocation. In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 1998. pp. 142–151. PLDI '98. ISBN 0897919874. Available from DOI 10.1145/277650.277714. Available from `https://doi.org/10.1145/277650.277714`.

WADLER, Philip. *The Expression Problem* [online]. 1998 [cit. 2023-06-19]. Available from `https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

WEGMAN, Mark N., and F. Kenneth ZADECK. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, apr, 1991, Vol. 13, No. 2, pp. 181–210. ISSN 0164-0925. Available from DOI 10.1145/103135.103136. Available from `https://doi.org/10.1145/103135.103136`.

WILSON, Tom, Gary GREWAL, Ben HALLEY, and Dilip BANERJI. An Integrated Approach to Retargetable Code Generation. In: *Proceedings of the 7th International Symposium on High-Level Synthesis*. Washington, DC, USA: IEEE Computer Society Press, 1994. pp. 70–75. ISSS '94. ISBN 0818657855.

WIMMER, Christian, and Michael FRANZ. Linear Scan Register Allocation on SSA Form. In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. New York, NY, USA: Association for Computing Machinery, 2010. pp. 170–179. CGO '10. ISBN 9781605586359. Available from DOI 10.1145/1772954.1772979. Available from `https://doi.org/10.1145/1772954.1772979`.

WULF, William Allan, Richard K. JOHNSSON, Charles B. WEINSTOCK, Steven O. HOBBS, and Charles M. GESCHKE. *The Design of an Optimizing Compiler*. USA: Elsevier Science Inc., 1975. ISBN 0444001581.

Appendix A
Acronyms

ALU	■	Arithmetic logic unit
AST	■	Abstract syntax tree
CFG	■	Control-flow graph
CISC	■	Complex instruction set computer
CRT	■	C runtime
CSE	■	common subexpression elimination
DAG	■	Directed acyclic graph
ELF	■	Executable and Linkable Format
GCC	■	GNU compiler collection
IR	■	Intermediate representation
IRC	■	Iterated register coalescing [George et al., 1996]
JIT	■	Just-in-time (compiler)
OS	■	Operating system
PE	■	Portable Executable
RISC	■	Reduced instruction set computer
SSA	■	Static single assignment

Appendix B
Contents of the electronic attachment

The electronic attachment contains the following directories:

```
/
├── benchmarks ........................ directory with benchmarks
│   ├── out ........................... directory with benchmark results
│   └── run_benchmarks.sh ............. script for running the benchmarks
├── examples .......................... directory with examples
├── ref ............................... a folder with reference outputs of tests
├── src ............................... directory with C sources
├── tests ............................. a folder with test files
├── text .............................. source code of the text of the thesis
│   ├── figures ....................... figures used in the thesis
│   ├── vlasami6-dip.pdf .............. text of the thesis in the PDF format
│   ├── vlasami6-dip.tex .............. TeX source of the thesis
│   └── vlasami6-dip.bib .............. bibliography in BibTeX
├── meson.build ....................... configuration for the Meson[1] build system
├── README.md ......................... electronic attachment description
└── test.sh ........................... testing script
```