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
Katedra teoretické informatiky

Master's Thesis

x86-64 nativní backend pro TinyC

Michal Vlasák

XX. XX. TODO
Supervisor: Ing. Petr Máj

Draft: 29. 3. 2023

/ Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V XX dne XX. XX. TODO

....

Abstrakt / Abstract

Klíčová slova: TeX, PDF, interaktivní prvky, multimédia, audio, video, 3D

Keywords: TeX, PDF, interactive elements, multimedia, audio, video, 3D

/ Contents

Draft: 29. 3. 2023

Chapter 1
Introduction

[ragalloc]

Chapter 2
Register allocation

This chapter describes the last of the three big conceptual parts of a usual compiler backend—the register allocation phase. Motivation, importance and possible approaches are introduced.

The x86-64 architecture (see [??]) 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 challanges 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.

2.1 Motivation

Most CPU architectures these days are register based. That means that interface of the CPU consists of a fixed number of registers and instructions that allow operations on these registers. For example registers may be 8 32-bit storage slots and instructions may allow performing arithmetic on these registers or allow loading/storing contents of register from/into memory. Memory is still an important part of these architectures—computations can't possibly fit all into a fixed number of registers of fixed size, it is the memory that allows us to store large amounts of data.

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. And not only that architectures these days usually have at least two classes of registers—general purpose registers and floating point registers...

After instruction selection (which tells us what instructions to use) and instruction scheduling (which refines the order these instructions are executed in), 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” or ”temporaries”. It is the goal of register allocation to transform the code so that real registers are used. The snippet corresponds to this middle end IR:

```
v1 = add 1, 2
```

The translation given above is suboptimal and in a reasonable compiler such code wouldn't get as far as to the register allocator: the middle end could fold the addition of constants into a constant, or the instruction selector could take at least take advantage of ”register plus immediate” instruction for addition. Nonetheless the example serves us well for showing how a simple allocation of registers might look like. 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`. Thes is the core idea of register allocation, we only need to store such values that will be needed in the future, and contraty to SSA we can use that to ”recycle” 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 (other than settings flags TODO). But at least in this case it is safely possible to remove that instruction. 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).

  But a question remains, can the register allocator remove the instruction? Or should it even? + Spill insertion.

As we have seen, opportunities often arise for register reuse. Then what are the situations where we can get out of registers? Well, when we can't get away with reuse. We should consider the difference between (naive, but illustrative) compilation of expression `1 + 2 + 3 + 4` and its possible register allocation:

```
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
```

And as opposed to that the ”right associative” version: `1 + (2 + (3 + 4))`

```
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
```

Even though both versions use the same amount of *virtual registers*, their opportunities to reuse registers and hence use of *physical registers* differs considerably. While the example is trivial and the right associative version could be transformed into the left associative version the highlighted issue here is, that even single benign expressions can use a surprising number of registers. Just extending the addition to more than 16 addends would leave us out of registers. So not only for variables (which for example in C nominally reside in memory), but even for temporaries we have to account for the fact that we simply can get out of registers.

References

Requests for correction