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
Katedra teoretické informatiky

Master's Thesis

x86-64 nativní backend pro TinyC

Michal Vlasák

XX. XX. TODO
Supervisor: Ing. Petr Máj

Draft: 20. 4. 2023

Draft: 20. 4. 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: TODO

Keywords: TODO

/ Contents

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.

2.2 Spilling

We can get out of registers, but we still have to store our (say intermediate) values *somewhere*. The solution is to put the values into memory. Resorting to putting the values to memory is called „spilling“. The system stack („frame stack“, „call stack“) is best suited for spilled values, much for the same reasons it is the best place for storing local variables: recursion is handled naturally and well, since each *invocation* of a function gets its own locations for storing the values—by storing the values on the stack, we allow functions to be *reentrant*.

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 (say smaller than 16 bits), also a feature efficiently supported by all common architectures these days. On the other hand accessing static slots of memory would pose similar challanges as global variables have - they may not be reachable with small relative offsets from any register and especially on 64 bits systems their absolute address is not only too large to be stored in literals in code, but using absolute addresses is also discouraged, because it means that the code becomes *position dependent*—change of the absolute address through a change in position in address space, would mean that our code would no longer be correct. Such absolute references require run time fix ups (called „relocations“) by the dynamic linker, which prevents reuse of the code between multiple processes.

[sec:use-of-spilled]

2.2.1 Using spilled values

As established, we will refer to spilled values through relative addressing (using either the stack pointer or the „base“ pointer, also often called the „frame pointer“, see TODO). But we have to get back to the problem at hand—we store values in registers, because we want to use the values in computations and processors (mostly) operate on registers. But now that we can't fit all values into registers and want to store them in memory, how do we use the values from memory and how do they even get there?

RISC processors usually employ a „load-store“ architecture. There are only a few dedicated instructions for reading values from memory into registers (`load` instructions) and writing register contents into memory (`store` instructions). All other instructions operate only with registers. This means that to even have a value to spill, it has to be in a register! Also only from that register, we can store the value to memory and later retrieve it, but again only into a register. The important observation here is, that resorting to storing the values in memory doesn't in general mean that we get around of doing register allocation. What we gain is that instead of having to store a value in a register in a long part of the code (because it is needed in all parts of that code), by putting the value into memory and retrieving it immediately before each use, we have made the register allocation problem simpler—a register is needed in more but smaller parts of the code. This means that a single machine register can be reused for several valuse, instead of being blocked by "one".

At least the register used for the store, doesn't have to be the same one used for the load. So even though we are constrained by the fact that registers have to be used for spills, we are not too constrained in choosing the registers for performing spills. Loads and stores inserted for handling spills are usually called the *spill code*.

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 instructions which can e.g. 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 reamins, 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.

An interesting perspective is, that 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 microoperations based on the load-store architecture, using some internal „architectural“ registers (not normally 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 „general purpose“ 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, let's say that `t4` needs to be spilled in the following:

```
add t3, t2
```

Instead of adding a load before the instruction of `t4` and a store of `t4` after, like this:

```
mov t4, [rbp+t3]
```

```
add t4, t2
mov [rbp+t3], t4
```

We can just operate on the memory location:

```
add [rbp+t3], t4
```

This example also showed, that as mentioned, at least with a very narrow local view, and with the first simpler solution, spilling `t3` doesn't help with the use of registers! We needed to introduce another pseudoregister, `t4`, to hold the value of `t3` loaded from memory. Indeed, spilling helps only in a broader scope, where splitting definitions / uses of one pseudoregister into single definitions and singe uses of many different pseudoregisters help making the register allocation solvable (due to limited number of registers) or easier.

Thus for spills, we want to introduce a new temporary for each new load and store, but that is not the case in the example, `t4` is used also for storing spilled `t3` back to memory. This is needed, since as we have discussed in TODO, most instructions on the x86-64 use the "two address code", where one of the operands is also the destination for the result. So it is not directly possible to store the result in a different location than the location of the first operand. Judging the snippet only locally, the transformation is fine, since the `original` value of `t3` was also „destroyed“ in the original version. In general, the original value of `t3`, before the addition, *might* be needed even after the addition. That is why the code generator might need to output the following sequence to add `t1` and `t2` into `t3` while preserving all of `t1`, `t2` and `t3`:

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

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

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

The code generator hoped that by assigning `t1` and `t3` the same register, the move instruction could be eliminitaed. 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 stored back into memory. Use of `t3`'s memory location as the first operand of the `add` instruction helps as before:

```
mov [rbp+t3], t1
add [rbp+t3], t2
```

But in some sense, this is a red herring—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. Alternatively instead of using memory location for the `add` instruction we could move the value from `t1` directly into memory, because store/load pair didn't have any meaning at all—we store the result of addition to `t3` in the last instruction afterall.

```
mov t4, t1
add t4, t2
mov [rbp+t3], 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. While the first example is two instructions, it is X bytes, while the second one is 3 instructions and Y bytes TODO: compare lengths of instructions. But in fact this is how we started in the first place—just have the values in the registers. The important difference is, that the register `t4` introduced by spilling, is a temporary, it's value ends right at the store to `t3`'s location in memory. Thus it makes the register allocation problem much easier than previously just with `t3`. For example now it is even more likely that `t4` and `t1` are assigned the same register, thus making the first instruction a noop. The reason why it is easier to merge with `t4` instead of `t3` is not really apparent from this local view, but we have TODO less interference.

Another point of view on the issue is, that essentially, in (the last snippet) in the first two instructions `t3` is represented by `t4`, while in the last instruction it shifts from being represented by `t4` to be represented by `t3`, which due to some previous decision, resides in a memory location. We have essentially split the register `t3` into multiple registers connected by moves and it improved the generated code. 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" (also called *coalescing*), and „splitting“ can both 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 as adding these contrasting notions makes great register allocation an even harder problem.

2.2.2 Interaction with instruction selection

As we have seen, the process of spilling needs to insert new instructions which together form the so colled "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 better thought than just load from memory and move to memory. But both of these problems—selecting the instructions to use for operations and additionally making nontrivial transformations based on the code at hand was exactly the job of instruction selection.

In principle, register allocator shouldn't care about the instructions. It should only cares about their effects on the registers. The result of register allocation should be the assignment of register to pseudoregisters. 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:

- Insert spill code in the register allocator pass.
- Return the list of spilled pseudoregisters, 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 the other hand, register allocation is already in the "backend", 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.

This can be application of that idea. On the other hand, as mentioned previously, machine independent representations of machine depednent instructions are possible, thus spill code insertion *could* also be made machine independent similarly as instruction selection.

The second makes the register allocation process pure in a sense. The register allocator never modifies the input, its result is a either a mapping of pseudoregisters to registers, or a list of pseudoregisters 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" backend 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 efter 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 (2.2.1).

The obvios 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 would want self contained and final results in possibly just one pass. The approaches used for spill code insertion are usually very connected to the core principle of the register allocation algorithm at hande, and choices of some approaches are discussed in the section 2.4.

2.3 Formalization

The terms used in the previous sections about register allocation were not properly defined, and perhaps even not commonly used for the concepts. The intention was to practically show the problems register allocation tries to solve and what needs to do to solve it, which of course includes optimizations that try to make the register allocation process more optimal.

One thing that has to be noted is the name „register allocation“ itself. We infact mentioned that register allocation is meant to map pseudoregisters to real machine 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 pseudoregister can be assigned a register. 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.* The assignments follows allocation—now that we can map every pseudoregister left into 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 the attacking each of those issues separately to reach more optimal results. But this distinction is of course not definitive.

2.3.1 Liveness, inteference

2.3.2 Live ranges

vs value vs variable.

2.3.3 Coalescing

2.3.4 Live range splitting

[sec:regalloc-techniques]

2.4 Techniques

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

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

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

  Most of the code compiled ahead of time can usually benefit from

2.4.1 Top-down, bottom-up

2.4.2 Backward

2.4.3 Linear scan

2.4.4 Graph coloring

2.4.5 Reduction (to another NP-complete problem and using a solver)

ILP, IBQP

2.4.6 SSA register allocation

References

Requests for correction