Lecture 18: Low-level code generation

David Hovemeyer

November 2, 2022

601.428/628 Compilers and Interpreters

High-level code generation

These slides present a few thoughts/recommendations about low-level (x86-64) code generation

Print comments as debugging output

Printing C-style `/* comments */` is a really useful way to emit debugging information in a way that won't interfere with the generated code being assembled

For example:

- Storage allocation decisions
- Computations involving memory layout

This technique is also useful for high-level code generation

Accessing memory in the stack frame

- Provided translations of `enter` and `leave` will (hopefully!) create ABI-compliant stack frames
- Assume that N bytes of memory are reserved in the stack frame
- `%rbp`$-N$ points to the "bottom" of the local memory area
- Assume that i is the offset of a local variable: it's displacement from `%rbp` should be $N - i$
- For example, if $N = 32$ and $i = 0$, use `-32(%rbp)` to access the memory location

Allocating storage for virtual registers

- Each function will use a certain number of virtual registers as
  - Storage for temporary (computed) values
  - Storage for (some) scalar local variables
- Note that `vr0` really means `%rax` and `vr1`, `vr2`, etc. are argument registers (`%rdi`, `%rsi`, etc.)
- For Assignment 4: allocate each vreg (other than `vr0` through `vr9`) in memory in the stack frame
  - This is in addition to the memory needed for local variables whose storage is in memory
- Assignment 5: you can do local register allocation to promote some virtual registers to CPU registers

Machine register sizes

- Each machine register has “subregisters” of various sizes
- These are specified as `Operand::Kind` values
- E.g., `Operand(Operand::MREG32, MREG_RAX)` represents the `%eax` register (i.e., the 32-bit sub-register of `%rax`)
- The `select_mreg_kind` helper function assists in selecting the correct machine register size

Instruction variants

- For instructions which move, compute, or compare values, there are different variants for different operand sizes
- The `select_ll_opcode` assists in selecting the correct low-level opcode

Temporary machine registers

- You can use `%r10` and `%r11` (and sub-registers of `%r10` and `%r11`) to store temporary values
- Use for dealing with situations such as
  - An x86-64 instruction can have at most one memory operand
  - Some instructions doesn't allow an immediate operand and a memory operand

What if a virtual register is used as a pointer?

Your high-level code will probably have operands like `(vr10)`, where a virtual register (in this case `vr10`) is being used as a pointer to access a data value in memory

Since virtual register values will be stored in memory, just referring to the contents of the virtual register requires a memory reference (e.g., `-24(%rbp)`). How to dereference a pointer if the pointer is in memory?

Solution: copy the pointer to a machine register, e.g.:

```
movq -24(%rbp), %r11
...code can now use (%r11) to dereference the pointer...
```

Conditions/decisions

- The comparison instructions provided high-level opcodes yield a boolean data value
- The `cjmp_t` and `cjmp_f` instructions consume this computed boolean data value
- How to generate code?
  - Use `set`*xx* instruction to use condition codes to set a boolean value in an 8 bit register

Example of evaluating a condition, control flow

```
...code for lhs subexpression...
...code for rhs subexpression...

cmpl rhsval, lhsval
setl %r10b
cmpb $0, %r10b
je .Lsome_label
```