Compiler Design - Cheatsheet

Compiler Structure

Simplified Compiler Structure

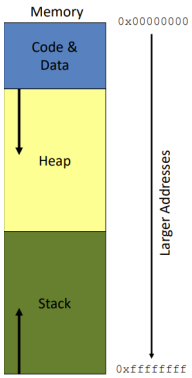
Source Code $\rightarrow$

$$\underbrace{\text{Lexical Analysis} \xrightarrow{\text{Token Stream}} \text{Parsing} \xrightarrow{\text{Abstract Syntax Tree}}}_{\text{Front End}}$$

$$\underbrace{\text{Intermediate Code Generation} \xrightarrow{\text{Intermediate Code}}}_{\text{Middle End}}$$

$$\underbrace{\text{Code Generation}}_{\text{Back End}}$$

Stack layout



Suffiexes

1. q = quadword (4 words)
2. l = long (2 words)
3. w = word (16-bit)
4. b = byte (8-bit)

movq-instruction

movq *SRC*, *DST* = *DST* $\leftarrow$ *SRC*

Flags

e (equality)	ZF is set
ne (inequality)	(not ZF)
g (greater than)	(not ZF) and (SF = OF)
l (less than)	SF <> OF
ge (greater or equal)	(SF = OF)
e (less than or equal)	SF <> OF or Z

Addressing

−8 (%rsp)	rsp − 8
(%rax, %rcx)	rax + 8·rcx
8(%rax, %rcx)	rax + 8·rcx + 8

leaq vs. movq

In leaq we just compute the address, in movq we dereference the address.

Callee vs. Caller saved

Caller saved register (Rdi, Rsi, Rdx, Rcx, R09, R08, Rax, R10, R11) are saved by the caller befor calling the function. Callee saved register (Rbx, R12, R13, R14, R15) are saved by the called function.

Parameter

1. 1 . . . 6: rdi, rsi, rdx, rcx, r8, r9
2. 7+: on the stack (in right-to-left order), nth arg. $((n-7)+2)\cdot 8 + rbp$

Why Intermediate Representations?

1. resulting code quality is poor (direct translation)
2. Richer source language features are hard to encode (Structured data types, Objects, . . .)
3. hard to optimize
4. Control-flow is not structured

Basic Blocks

1. Starts with a label that names the entry point of the basic block
2. Ends with a control-flow instruction
3. Contains no other control-flow instructions
4. Contains no interior label used as a jump target

CFGs

1. Nodes are basic blocks
2. There is a directed edge from node A to node B if the control flow instruction at the end of block A might jump to the label of block B
3. No two blocks have the same label

getelementptr instruction

The first argument is always a type used as the basis for the calculations. The second argument is always a pointer or a vector of pointers, and is the base address to start from. The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed. GEP never dereferences the address it's calculating.

```
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};
```

```
int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

is translated in:

```
%arrayidx = getelementptr %struct.ST, ptr %s, i64 1,
i32 2, i32 1, i64 5, i64 13
ret ptr %arrayidx
```