Where We Are

Source code

if (b == 0) a = b;

Low-level IR code

Optimized Low-level IR code

Lexical, Syntax, and Semantic Analysis
IR Generation

Optimizations

Assembly code generation

Assembly code

cmp $0,%rcx

cmovz %rax,%rdx
Code Generation: IR to Assembly Translation

• IR code (TAC):
  • Variables (and temporaries)
  • No run-time stack
  • No calling sequences
  • Abstract set of instructions

• Translation to x86-64:
  • Calling sequences:
    • Translate function calls and returns
    • Manage run-time stack
  • Variables:
    • globals, locals, arguments, etc. assigned memory location
  • Instruction selection:
    • map sets of low level IR instructions to instructions in the target machine

\[
\begin{align*}
t3 &= a \cdot x \\
t3 &= t2 \times t3 \\
t0 &= t1 + t2 \\
r &= t0 \\
t4 &= w + 1
\end{align*}
\]
Big Picture: Memory Layout

- Stack variables
  - Param \( n \)
    - \( \ldots \)
    - Param 0
  - Return address
  - Saved regs
  - Local 0
    - \( \ldots \)
    - Local \( n \)

- Heap variables

- Global variables
  - Global \( n \)
    - \( \ldots \)
  - Global 0
x86-64 stack frames

x86-64/Linux ABI
No base pointer
1st 6 args in registers
Stack access relative to %rsp
Compiler knows frame size
16-byte-aligned frames
x86-64 with all arguments on the stack

All args on stack
Consistent, easy
Use redzone for arg prep

Stack pointer `%rsp`

High addresses

Caller Frame

Low addresses

Callee Frame

Stack Top

Arguments for next call

Saved Registers + Local Variables

Callee Argument 0

Callee Argument n
x86-64/Linux ABI: register conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

Only `%rsp` is special-purpose.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
roostc calling convention

• Always follow x86-64/Linux register save convention.

• To interface with external code (LIB), use:
  • x86-64/Linux calling convention.

• To interface with other roostc-generated code, use one of:
  1. frame pointer and stack pointer, all args on stack
     • Easy, more work to convert if you convert to #3 later.
  2. stack pointer only, all args on stack (recommended)
     • Moderately easy, easier to convert to #3 later.
  3. x86-64/Linux calling convention
     • Harder, requires more register allocation work, more efficient, only use this later if you have time.
Simple stack frame maintenance: stack args

• Compute offsets for each variable
  • Count args, local vars, temporaries
  • Assign each an offset relative to %rsp (top of stack frame).
  • Args are fixed, everything else flexible.

• Avoid using callee-save registers – no saving required.

• Prologue at start of function:
  \texttt{\textbf{subq} $24$, \%rsp}  \# space for 3 locals (or 2 plus alignment)

• Epilogue/cleanup at end of function:
  \texttt{\textbf{addq} $24$, \%rsp}
  \texttt{\textbf{retq}}

• Many improvements possible, but a good starting point.
Simple template-based code generation

\[ a = p + q \quad \rightarrow \quad \text{movq 16(\%rsp), \%rax} \]
\[ \quad \text{addq 8(\%rsp), \%rax} \]
\[ \quad \text{movq \%rax, 24(\%rsp)} \]

• Need to consider many language constructs:
  • Operations: arithmetic, logic, comparisons
  • Accesses to local variables, global variables
  • Array accesses, field accesses
  • Control flow: conditional and unconditional jumps
  • Method calls, dynamic dispatch
  • Dynamic allocation (new)
  • Run-time checks
Division

movq ..., %rcx # divisor, can't be %rax or %rdx
movq ..., %rax # dividend
cqto # sign-extend %rax into %rdx:%rax
idivq %rcx # divide %rdx:%rax by %rcx
    # quotient in %rax
    # remainder in %rdx
String Literals

.rodata
...
.align 8
.quad 13
strlit3:
.ascii "Hello, World!"
...
.text
...

# t4 = "Hello, World!"
# Works on both LLVM/macOS and GCC/Linux:
    leaq strlit3(%rip), %rax    # GCC only: movq $strlit3, %rax
    movq %rax, 8(%rsp)
# Roost.println(t4);
# external call must use x86-64/Linux ABI calling convention
    movq 8(%rsp), %rdi     # load t4 as arg 1
    callq __LIB_println

Method vectors/vtables and vtable pointer initialization will be similar.
cmpq and testq

cmpq %rcx,%rax computes %rax - %rcx, sets CF, OF SF, ZF, discards result

testq %rax,%rcx computes %rax & %rcx, sets SF, ZF, discards result

Flags/condition codes:
- CF: carry flag, 1 iff carry out
- OF: overflow flag, 1 iff signed overflow
- SF: sign flag, 1 iff result's MSB=1
- ZF: zero flag, 1 iff result=0

Common pattern to test for 0 or <0: testq %rax, %rax
### jmp and j\textit{cc}

<table>
<thead>
<tr>
<th>j\textit{cc}</th>
<th>Condition</th>
<th>Jump iff ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je, jz</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne, jnz</td>
<td>\sim ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>jg</td>
<td>\sim(SF\land OF) \land \sim ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>\sim(SF\land OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF\land OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF\land OF) \land ZF</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>\sim SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>ja</td>
<td>\sim CF \land \sim ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

**Always jump**

**Jump iff condition**
Set has all the same flavors as conditional jump.
Accessing Heap Data

• Heap data allocated with new (Java) or malloc (C/C++)
  • Allocation function returns address of allocated heap data
  • Access heap data through that reference

• Array accesses in Java
  • access \( a[i] \) requires:
    • computing address of element: \( a + i \times \text{size} \)
    • accessing memory at that address
  • Example:
    • assume size of array elements is 8 bytes, and local variables \( a, i \) (offsets –8, -16)

\[
\text{a}[i] = 1 \quad \Rightarrow \quad \text{mov } -8(\%\text{rbp}), \%\text{rdx} \quad \text{(load a)} \\
\text{mov } -16(\%\text{rbp}), \%\text{rcx} \quad \text{(load i)} \\
\text{mov } $1, (\%\text{rdx,}\%\text{rcx,8}) \quad \text{(store into the heap)}
\]
Run-time Checks

• Check if array/object references are non-null
• Check if array index is within bounds
  • if v holds the address of an array, insert array bounds checking code for v before each load _ = v[i] or store v[i] = _
  • Array length is stored just before array elements:

```
cmp $0, -24(%rbp)     (compare i to 0)
jl  ArrayBoundsError23 (test lower bound)
mov -16(%rbp), %rcx   (load v into %ecx)
mov -8(%rcx), %rcx    (load array length into %ecx)
cmp -24(%rbp), %rcx   (compare i to array length)
jle ArrayBoundsError23 (test upper bound)
...```
Field Offsets

- Offsets of fields from beginning of object known statically

```c
struct Shape {
    field ll : Point /* 8 */
    field ur : Point /* 16 */
}
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ll</td>
<td>Point</td>
</tr>
<tr>
<td>ur</td>
<td>Point</td>
</tr>
</tbody>
</table>
Method Arguments

- Receiver is (implicit) argument to method

```rust
struct A {
    method f(x: i64,
              y: i64 ) -> unit {
    }
}
```

compile as

```rust
fn f(self: A,
     x: i64,
     y: i64) -> unit {
}
```
Code Generation: Calls

• **Pre-function-call code:**
  • Save registers
  • Push parameters
  • call function by its label

• **Pre-method call:**
  • Save registers
  • Push parameters
  • *Push receiver object reference*
  • *Lookup method in vtable*
Example call: using declared fn name

```rust
fn foo(x: i64, y:i64) -> i64 { ... }
```

```
... foo(2,3)
```

```assembly
movq $3, -8(%rsp)
movq $2, -16(%rsp)
subq $16, %rsp
callq foo
addq $16, %rsp
```

Direct call because function name is known statically.

Uses redzone to setup stack args instead of push.
Example call: using expression of fn type

```rust
def fn foo(x: i64, y: i64) -> i64 { ... }
...
let f:(x: i64, y: i64) -> i64 = foo in {
    f(2, 3)
    movq $3, -8(%rsp)
    movq $2, -16(%rsp)
    movq 24(%rsp), %rax
    subq $16, %rsp
    callq *%rax
    addq $16, %rsp
}
```

Indirect call because function address is only known dynamically.
Code Generation: Library Calls

• Use x86-64/Linux ABI calling convention

• Warning: library functions may modify caller save registers

```assembly
# Roost.printi64(301)
movq $301, %rdi
callq __LIB_printi64

# Roost.random(301)
movq $301, %rdi
call __LIB_random
movq %rax, -32(%rbp)
```
Code Generation: Allocation

• Heap allocation: o = new C()
  • Allocate heap space for object

```
 movq  $24, %rdi  # 3 fields
 callq __LIB_allocateObject
 # reference in %rax
```