Code Generation
Roadmap

Last time, we learned about variable access
- Local vs. global variables
- Static vs. dynamic scopes

Today
- We’ll start getting into the details of MIPS
- Code generation
Roadmap

Scanner

Parser

Tokens

Parse Tree

AST

Static-Semantic Analysis

Annotated AST

Symbol Table

IR Codegen

Optimizer

MC Codegen

Backend
The Compiler Back End

Unlike in the front end, we can skip phases without sacrificing correctness

Actually have a couple of options:

– What phases do we do?
– How do we order our phases?
Outline

Possible compiler designs

– Generate IR code or machine-code code directly?
– Generate during SDT or as another phase?

Diagram:

- Frontend
  - IR Codegen
    - Optimizer
      - MC Codegen
  - MC Codegen
How Many Passes Do We Want?

Fewer passes
  – Faster compiling
  – Less storage required
  – May increase burden on programmer

More passes
  – Heavyweight
  – Can lead to better modularity
To Generate IR Code or Not?

Generate Intermediate Representation:
– More amenable to optimization
– More flexible output options
– Can reduce the complexity of code generation

Go straight to machine code:
– Much faster to generate code (skip 1 pass, at least)
– Less engineering in the compiler
What Might the IR Do?

Provide illusion of infinitely many registers
“Flatten out” expressions
– Does not allow building up complex expressions

3AC (Three-Address Code)
– Instruction set for a fictional machine
– Every operator has at most 3 operands
3AC Example

\begin{align*}
\text{if } (x + y \cdot z > x \cdot y + z) & \\
& \quad a = 0; \\
& \quad b = 2;
\end{align*}

\begin{align*}
\text{tmp1} & = y \cdot z \\
\text{tmp2} & = x + \text{tmp1} \\
\text{tmp3} & = x \cdot y \\
\text{tmp4} & = \text{tmp3} + z \\
\text{if } (\text{tmp2} \leq \text{tmp4}) & \text{ goto } L \\
& \quad a = 0 \\
L: & \quad b = 2
\end{align*}
3AC Instruction Set

Assignment
- \( x = y \text{ op } z \)
- \( x = \text{ op } y \)
- \( x = y \)

Jumps
- \( \text{if } ( x \text{ op } y) \text{ goto } L \)

Indirection
- \( x = y[z] \)
- \( y[z] = x \)
- \( x = &y \)
- \( x = *y \)
- \( *y = x \)

Call/Return
- \( \text{param } x,k \)
- \( \text{retval } x \)
- \( \text{call } p \)
- \( \text{enter } p \)
- \( \text{leave } p \)
- \( \text{return } \)
- \( \text{retrieve } x \)

Type Conversion
- \( x = \text{AtoB } y \)

Labeling
- \( \text{label } L \)

Basic Math
- \( \text{times, plus, etc.} \)
3AC Representation

Each instruction represented using a structure called a “quad”

- Space for the operator
- Space for each operand
- Pointer to auxiliary info
  - Label, successor quad, etc.

Chain of quads sent to an architecture-specific machine-code-generation phase
Direct Machine-Code Generation

Option 1
– Have a chain of quad-like structures where each element is a machine-code instruction
– Pass the chain to a phase that writes to file

Option 2
– Write code directly to the file
– Greatly aided by the assembler’s capabilities
– Assembler allows us to use function names, labels in output
Moo: Skip Building a Separate IR

Generate code (of a very simple kind) by traversing the AST

– Add codeGen methods to the AST nodes
– Directly emit corresponding code into file
Correctness/Efficiency Tradeoffs

Two high-level goals

1. Generate correct code
2. Generate efficient code

It can be difficult to achieve both of these at the same time

– Why?
A Simplified Strategy

Make sure we don’t have to worry about running out of registers

– For each operation (built-in, like plus, or user-defined, like a call on a user-define function), we’ll put all arguments on the stack

– We’ll make liberal use of the stack for computation

– We’ll make use of only two registers

  • Only use $t1 and $t0 for computation
The CodeGen Pass

We’ll now go through a high-level idea of how the topmost nodes in the program are generated.
The Responsibility of Different Nodes

Many nodes simply “direct traffic”

– ProgramNode.codeGen
  • call codeGen on the child

– List-node types
  • call codeGen on each element in turn

– DeclNode
  • StructDeclNode – no code to generate!
  • FnDeclNode – generate function body
  • VarDeclNode – varies on context! Globals vs. locals
Generating a Global-Variable Declaration

Source code:

```c
int name;
struct MyStruct instance;
```

In `varDeclNode`

Generate:

```assembly
.data
.align 2  #Align on word boundaries
_name: .space N  #(N is the size of variable)
```
Generating a Global-Variable Declaration

.data
.align 2  #Align on word boundaries
_name: .space N  #(N is the size of variable)

How do we know the size?
– For scalars, well-defined: int, bool (4 bytes)
– structs, 4 * size of the struct
We can calculate this during name analysis
Generating Function Definitions

Need to generate

– Preamble
  • Sort of like the function signature
– Prologue
  • Set up the function’s AR
– Body
  • Code to perform the computation
– Epilogue
  • Tear down the function’s AR
### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>return address</td>
</tr>
<tr>
<td>$v0</td>
<td>used for system calls and to return int values from function calls, including the syscall that reads an int</td>
</tr>
<tr>
<td>$f0</td>
<td>used to return double values from function calls, including the syscall that reads a double</td>
</tr>
<tr>
<td>$a0</td>
<td>used for output of int and string values</td>
</tr>
<tr>
<td>$f12</td>
<td>used for output of double values</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>temporaries for ints</td>
</tr>
<tr>
<td>$f0 - $f30</td>
<td>registers for doubles (used in pairs; i.e., use $f0 for the pair $f0, $f1)</td>
</tr>
</tbody>
</table>

Also $LO and $HI, special-purpose registers used by multiplication and division instructions.
Program Structure

Data
- Label: .data
- Variable names & size; heap storage

Code
- Label: .text
- Program instructions
- Starting location: main
- Ending location

For the main function, generate:
```
.text
.globl main
main:
```

For all other functions, generate:
```
.text
_<functionName>:
```
Data

name: type value(s)

- E.g.
  - v1: .word 10
  - a1: .byte ‘a’, ’b’
  - a2: .space 40
    - 40 here is allocated space – no value is initialized
Memory Instructions

`lw  register_destination, RAM_source`
– copy word (4 bytes) at source RAM location to destination register.

`lb  register_destination, RAM_source`
– copy byte at source RAM location to low-order byte of destination register

`li  register_destination, value`
– load immediate value into destination register
Memory Instructions

sw  register_source, RAM_dest
    – store word in source register into RAM destination

sb  register_source, RAM_dest
    – store byte in source register into RAM destination
# Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$t0,$t1,$t2</td>
</tr>
<tr>
<td>sub</td>
<td>$t2,$t3,$t4</td>
</tr>
<tr>
<td>addi</td>
<td>$t2,$t3, 5</td>
</tr>
<tr>
<td>addu</td>
<td>$t1,$t6,$t7</td>
</tr>
<tr>
<td>subu</td>
<td>$t1,$t6,$t7</td>
</tr>
<tr>
<td>mult</td>
<td>$t3,$t4</td>
</tr>
<tr>
<td>div</td>
<td>$t5,$t6</td>
</tr>
<tr>
<td>mfhi</td>
<td>$t0</td>
</tr>
<tr>
<td>mflo</td>
<td>$t1</td>
</tr>
</tbody>
</table>
Control Instructions

Unconditional branch to target
- Specified as a relative transfer of control to target (i.e., target = IP + delta)
- IP implicit; delta is a 16-bit immediate operand (a signed 16-bit number)

Unconditional jump to target
- Specified as an absolute transfer of control to target
- Target limited to 26 bits

Indirect jump
- Specified as an absolute transfer of control to address in $t3

\[
\begin{align*}
\text{b} & \quad \text{target} \\
\text{beq} & \quad \text{t0, t1, target} \\
\text{blt} & \quad \text{t0, t1, target} \\
\text{ble} & \quad \text{t0, t1, target} \\
\text{bgt} & \quad \text{t0, t1, target} \\
\text{bge} & \quad \text{t0, t1, target} \\
\text{bne} & \quad \text{t0, t1, target} \\
\text{j} & \quad \text{target} \\
\text{jr} & \quad \text{t3} \\
\text{jal} & \quad \text{sub_label} \quad \# "jump and link"
\end{align*}
\]

Jump to sub_label, and store the return address in $ra
TODO

Watch ALL MIPS and SPIM tutorials online
– pages.cs.wisc.edu/~cs536-1/resources.html

MIPS tutorial
Roadmap

Today
– Talked about compiler back-end design points
– Decided to go directly from AST to machine code for our compiler

Next time:
– Run through what the actual codegen pass looks like