Code Generation
Roadmap

- Scanner
- Tokens
- Parser
- Parse Tree
- AST
- Semantic Analysis
- Annotated AST
- Symbol Table

Backend

- IR Codegen
- Optimizer
- MC Codegen
The Compiler Back-end

• Unlike 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 MC code directly?
  – Generate during SDT or as another phase?
How many passes do we want?

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

• More passes
  – Heavyweight
  – Can lead to better modularity
  – We’ll go with this approach for CATS
To Generate IR Code or Not?

• If we do generate an Intermediate Representation:
  – More amenable to optimization
  – More flexible output options
  – Can reduce the complexity of code generation

• If we 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?

• Infinite-register operations
• “Flatten out” expressions
  – Does not allow build-up of complex expressions
• 3AC (Three-Address Code)
  – Pseudocode-machine style instruction set
  – Every operator has at most 3 operands
3AC Example

if (x + y * z > x * y + z)
a = 0;
b = 2;

tmp1 = y * z
tmp2 = x+tmp1
tmp3 = x*y
tmp4 = tmp3+z
if (tmp2 <= tmp4) goto L
    a = 0
L: b = 2
3AC Instruction Set

• Assignment
  – $x = y \text{ op } z$
  – $x = \text{ op } y$
  – $x = y$

• Jumps
  – if $(x \text{ op } y)$ goto $L$

• Indirection
  – $x = y[z]$
  – $y[z] = x$
  – $x = \&y$
  – $x = \ast y$
  – $\ast y = x$

• Call/Return
  – param $x,k$
  – retval $x$
  – call $p$
  – enter $p$
  – leave $p$
  – return
  – retrieve $x$

• Type Conversion
  – $x = \text{ AtoB } y$

• Labeling
  – label $L$

• Basic Math
  – times, plus, etc.
3AC Representation

• Each instruction represented using a structure called a “quad”
  – Space for the operator
  – Space for each operand
  – Pointer to auxilary info
    • Label, succesor quad, etc.

• Chain of quads sent to an architecture specific MC codegen phase
3AC LLVM Example
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 assembly conventions here
      – Assembler allows us to use function names, labels in output
CATS: Skip the IR

• Traverse AST
  – add codeGen methods to the AST nodes
  – Directly spit 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?
Simplifying assumptions

• Make sure we don’t have to worry about running out of registers
  – We’ll put all function arguments on the stack
  – We’ll make liberal use of the stack for computation
    • 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 Effect of Different Nodes

- Many nodes simply structure their results
  - 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 v locals
Generating Global Variable Declaration

• **Source code:**

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

• **In varDeclNode**

  Generate:

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

```assembly
.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
  – Body
    • Do the thing
  – Epilogue
    • Tear down the function
# MIPS crash course

## 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>
Program structure

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

• Code
  – Label: .text
  – Program instructions
  – Starting location: main
  – Ending location
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
Mem 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
Mem 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

- **add** $t0,t1,t2$
- **sub** $t2,t3,t4$
- **addi** $t2,t3,5$
- **addu** $t1,t6,t7$
- **subu** $t1,t6,t7$

- **mult** $t3,t4$ Stores result in $LO$

- **div** $t5,t6$ Stores result in $LO$ and Remainder in $HI$

- **mfhi** $t0$
- **mflo** $t1$
Control instructions

b      target
beq    $t0,$t1,target
blt    $t0,$t1,target
ble    $t0,$t1,target
bgt    $t0,$t1,target
bge    $t0,$t1,target
bne    $t0,$t1,target

j      target
jr     $t3

jal    sub_label       # "jump and link"

Jump and store return address in $31
TODO

• Watch **ALL** MIPS and SPIM tutorials online
  – pages.cs.wisc.edu/~loris/cs536s16/resources.html

• MIPS tutorial