Code Generation

Lecture 12

Lecture Outline

• Topic 1: Basic Code Generation
  - The MIPS assembly language
  - A simple source language
  - Stack-machine implementation of the simple language

• Topic 2: Code Generation for Objects

From Stack Machines to MIPS

• The compiler generates code for a stack machine with accumulator

• We want to run the resulting code on the MIPS processor (or simulator)

• We simulate stack machine instructions using MIPS instructions and registers

Simulating a Stack Machine...

• The accumulator is kept in MIPS register $a0

• The stack is kept in memory
  - The stack grows towards lower addresses
  - Standard convention on the MIPS architecture

• The address of the next location on the stack is kept in MIPS register $sp
  - The top of the stack is at address $sp + 4

MIPS Assembly

MIPS architecture
  - Prototypical Reduced Instruction Set Computer (RISC) architecture
  - Arithmetic operations use registers for operands and results
  - Must use load and store instructions to use operands and results in memory
  - 32 general purpose registers (32 bits each)
    - We will use $sp, $a0 and $t1 (a temporary register)

• Read the SPIM documentation for details

A Sample of MIPS Instructions

- lw reg, offset(reg)
  - Load 32-bit word from address reg + offset into reg
- add reg, reg, reg
  - reg ← reg + reg
- sw reg, offset(reg)
  - Store 32-bit word in reg at address reg + offset
- addiu reg, reg, imm
  - reg ← reg + imm
  - "u" means overflow is not checked
- li reg imm
  - reg ← imm
MIPS Assembly. Example.

• The stack-machine code for $7 + 5$ in MIPS:
  
  acc ← 7
  push acc
  acc ← 5
  acc ← acc + top_of_stack
  pop

• We now generalize this to a simple language...

A Small Language

• A language with integers and integer operations

\[ P \rightarrow D; P \mid D \]
\[ D \rightarrow \text{def id(ARGS) = E;} \]
\[ \text{ARGS} \rightarrow \text{id, ARGs} \mid \text{id} \]
\[ E \rightarrow \text{int} \mid \text{id} \mid \text{if} \ E \]
\[ \mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1, \ldots, E_n) \]

A Small Language (Cont.)

• The first function definition $f$ is the “main” routine
• Running the program on input $i$ means computing $f(i)$
• Program for computing the Fibonacci numbers:
  
  \[
  \text{def fib}(x) = \begin{cases} 
  0 & \text{if } x = 1 \\
  1 & \text{if } x = 2 \\
  \text{fib}(x - 1) + \text{fib}(x - 2) & \text{else}
  \end{cases}
  \]

Code Generation Strategy

• For each expression $e$ we generate MIPS code that:
  - Computes the value of $e$ in $a0$
  - Preserves $sp$ and the contents of the stack

• We define a code generation function $cgen(e)$ whose result is the code generated for $e$

Code Generation for Constants

• The code to evaluate a constant simply copies it into the accumulator:
  
  $cgen(i) = \text{li } a0 \ i$

• This preserves the stack, as required

• Color key:
  - RED: compile time
  - BLUE: run time

Code Generation for Add

\[
\begin{align*}
\text{cgen}(e_1 + e_2) &= \text{cgen}(e_1) + \text{cgen}(e_2) \\
\text{sw } a0 0(\text{sp}) &\quad \text{print “sw } a0 0(\text{sp})” \\
\text{addiu } sp &sp -4 \quad \text{print “addiu } sp \ sp -4” \\
\text{cgen}(e_1) &\quad \text{cgen}(e_1) \\
\text{lw } t1 4(\text{sp}) &\quad \text{print “lw } t1 4(\text{sp})” \\
\text{add } a0 &t1 a0 \quad \text{print “add } a0 t1 a0” \\
\text{addiu } sp &sp 4 &\quad \text{print “addiu } sp \ sp 4”
\end{align*}
\]
Code Generation for Add. Wrong!

- Optimization: Put the result of $e_1$ directly in $t1$?
  \[
  \text{cgen}(e_1 + e_2) = \\
  \quad \text{cgen}(e_1) \\
  \quad \text{move } t1 \ x0 \\
  \quad \text{cgen}(e_2) \\
  \quad \text{add } x0 \ t1 \ x0
  \]

- Try to generate code for $3 + (7 + 5)$

Code Generation Notes

- The code for $+$ is a template with “holes” for code for evaluating $e_1$ and $e_2$

- Stack machine code generation is recursive
  - Code for $e_1 + e_2$ is code for $e_1$ and $e_2$ glued together

- Code generation can be written as a recursive-descent of the AST
  - At least for expressions

Code Generation for Sub and Constants

- New instruction: sub reg, reg, reg
  - Implements reg, \leftarrow reg, reg
  \[
  \text{cgen}(e_1 - e_2) = \\
  \quad \text{cgen}(e_1) \\
  \quad \text{sw } x0 \ O(sp) \\
  \quad \text{addiu } x0 \ sp \ -4 \\
  \quad \text{cgen}(e_2) \\
  \quad \text{lw } t1 \ 4(sp) \\
  \quad \text{sub } x0 \ t1 \ x0 \\
  \quad \text{addiu } sp \ sp \ 4
  \]

Code Generation for Conditional

- We need flow control instructions

- New instruction: beq reg, reg, label
  - Branch to label if reg, \leftarrow reg

- New instruction: b label
  - Unconditional jump to label

Code Generation for If (Cont.)

\[
\text{cgen}(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \\
\quad \text{cgen}(e_1) \\
\quad \text{sw } x0 \ O(sp) \\
\quad \text{addiu } x0 \ sp \ -4 \\
\quad \text{cgen}(e_2) \\
\quad \text{lw } t1 \ 4(sp) \\
\quad \text{addiu } sp \ sp \ 4 \\
\quad \text{beq } x0 \ t1 \ \text{true_branch}
\]

\[
\text{false_branch:} \\
\quad \text{cgen}(e_4) \\
\quad \text{addiu } x0 \ sp \ -4
\]

\[
\text{true_branch:} \\
\quad \text{cgen}(e_3) \\
\quad \text{beq } x0 \ t1 \ \text{true_branch}
\]

\[
\text{end_if:} \\
\quad \text{cgen}(e_4)
\]

The Activation Record

- Code for function calls and function definitions depends on the layout of the AR

- A very simple AR suffices for this language:
  - The result is always in the accumulator
  - No need to store the result in the AR
  - The activation record holds actual parameters
    - For $(x_1, \ldots, x_n)$ push $x_n, \ldots, x_1$ on the stack
    - These are the only variables in this language
The Activation Record (Cont.)

- The stack discipline guarantees that on function exit $sp$ is the same as it was on function entry
  - No need for a control link
- We need the return address
- A pointer to the current activation is useful
  - This pointer lives in register $fp$ (frame pointer)
  - Reason for frame pointer will be clear shortly

The Activation Record

- Summary: For this language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices
- Picture: Consider a call to $f(x,y)$, the AR is:

\[
\begin{array}{c}
\text{FP} \\
\text{old fp} \\
\text{y} \\
\text{x} \\
\text{SP}
\end{array}
\]

Code Generation for Function Call

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation
- New instruction: jal label
  - Jump to label, save address of next instruction in $ra$
  - On other architectures the return address is stored on the stack by the "call" instruction

Code Generation for Function Call (Cont.)

cgen(f(e_1, ..., e_n)) =

- The caller saves its value of the frame pointer
- Then it saves the actual parameters in reverse order
- The caller saves the return address in register $ra$
- The AR so far is $4*n+4$ bytes long

Code Generation for Function Definition

- New instruction: jr reg
  - Jump to address in register reg

cgen(def f(x_1, ..., x_n) = e) =

- Note: The frame pointer points to the top, not bottom of the frame
  - The callee pops the return address, the actual arguments and the saved value of the frame pointer
  - $z = 4*n + 8$

Calling Sequence: Example for $f(x,y)$

Before call | On entry | Before exit | After call
--- | --- | --- | ---

\[
\begin{array}{c}
\text{FP} \\
\text{old fp} \\
\text{y} \\
\text{x} \\
\text{SP}
\end{array}
\]
Code Generation for Variables

• Variable references are the last construct

• The “variables” of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller

• Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp$

Code Generation for Variables (Cont.)

• Example: For a function $f(x, y) = e$ the activation and frame pointer are set up as follows:

```
old fp
FP

\[x\]
\[\text{return}\]

\[\text{y}\]
\[\text{x}\]
\[\text{y}\text{ is at } fp + 4\]
\[\text{x}\text{ is at } fp + 8\]
```

Summary

• The activation record must be designed together with the code generator

• Code generation can be done by recursive traversal of the AST

• We recommend you use a stack machine for your Cool compiler (it’s simple)

An Improvement

• Idea: Keep temporaries in the AR

• The code generator must assign a location in the AR for each temporary
Example

```python
def fib(x):
    if x == 1:
        return 0
    elif x == 2:
        return 1
    else:
        return fib(x - 1) + fib(x - 2)
```

- What intermediate values are placed on the stack?
- How many slots are needed in the AR to hold these values?

How Many Temporaries?

- Let $NT(e)$ be the number of temporaries needed to evaluate $e$
- $NT(e_1 + e_2)$ needs at least as many temporaries as $NT(e_1)$
- $NT(e_1 + e_2)$ needs at least as many temporaries as $NT(e_2) + 1$
- Space used for temporaries in $e_1$ can be reused for temporaries in $e_2$

The Equations

- $NT(e_1 * e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \max(NT(e_1), NT(e_2), NT(e_3), NT(e_4))$
- $NT(id) = 0$
- $NT(int) = 0$

Is this bottom-up or top-down?
What is $NT(\text{code for fib...})$?

The Revised AR

- For a function definition $f(x_1,...,x_n) = e$ the AR has $2 + n + NT(e)$ elements
  - Return address
  - Frame pointer
  - $n$ arguments
  - $NT(e)$ locations for intermediate results

Revised Code Generation

- Code generation must know how many temporaries are in use at each point
  - Add a new argument to code generation: the position of the next available temporary
**Code Generation for + (original)**

\[
cgen(e_1 + e_2) = \\
\quad cgen(e_1) \\
\quad sw \ a0 \ 0($sp) \\
\quad addiu \ $sp \ $sp -4 \\
\quad cgen(e_2) \\
\quad lw \ $t1 \ 4($sp) \\
\quad add \ $a0 \ $t1 \ $a0 \\
\quad addiu \ $sp \ $sp \ 4
\]

**Code Generation for + (revised)**

\[
cgen(e_1 + e_2, nt) = \\
\quad cgen(e_1, nt) \\
\quad sw \ a0 \ nt($fp) \\
\quad addiu \ $sp \ nt + 4 \\
\quad cgen(e_2, nt + 4) \\
\quad lw \ $t1 \ nt($fp) \\
\quad add \ $a0 \ $t1 \ $a0
\]

**Notes**

- The temporary area is used like a small, fixed-size stack
- Exercise: Write out \( cgen \) for other constructs

**Object Layout**

- **OO implementation**: Stuff from last part + more stuff
- **OO Slogan**: If \( B \) is a subclass of \( A \), then an object of class \( B \) can be used wherever an object of class \( A \) is expected
- This means that code in class \( A \) works unmodified for an object of class \( B \)

**Two Issues**

- How are objects represented in memory?
- How is dynamic dispatch implemented?
Object Layout Example

Class A {
    a: Int <- 0;
    d: Int <- 1;
    f(): Int { a <- a + d; }
};

Class B inherits A {
    b: Int <- 2;
    f(): Int { a; }
    g(): Int { a <- a - b; }
};

Class C inherits A {
    c: Int <- 3;
    h(): Int { a <- a * c; }
};

Object Layout (Cont.)

• Attributes a and d are inherited by classes B and C

• All methods in all classes refer to a

• For A methods to work correctly in A, B, and C objects, attribute a must be in the same “place” in each object

Object Layout (Cont.)

An object is like a struct in C. The reference foo.field is an index into a foo struct at an offset corresponding to field

Objects in Cool are implemented similarly
- Objects are laid out in contiguous memory
- Each attribute stored at a fixed offset in object
- When a method is invoked, the object is self and the fields are the object’s attributes

Cool Object Layout

• The first 3 words of Cool objects contain header information:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Class Tag</td>
</tr>
<tr>
<td>4</td>
<td>Object Size</td>
</tr>
<tr>
<td>8</td>
<td>Dispatch Ptr</td>
</tr>
<tr>
<td>12</td>
<td>Attribute 1</td>
</tr>
<tr>
<td>16</td>
<td>Attribute 2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Subclasses

Observation: Given a layout for class A, a layout for subclass B can be defined by extending the layout of A with additional slots for the additional attributes of B

Leaves the layout of A unchanged (B is an extension)
Subclasses (Cont.)

- The offset for an attribute is the same in a class and all of its subclasses
  - Any method for an \( A_1 \) can be used on a subclass \( A_2 \)
  - Consider layout for \( A_n < \ldots < A_3 < A_2 < A_1 \)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Class</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Atag</td>
<td>5</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Btag</td>
<td>6</td>
<td>a</td>
<td>d</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Ctag</td>
<td>6</td>
<td>a</td>
<td>d</td>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic Dispatch

- Consider the following dispatches (using the same example)

Object Layout Example (Repeat)

Class A {
  a: Int <- 0;
  d: Int <- 1;
  f(): Int { a <- a + d };
}

Class B inherits A {
  b: Int <- 2;
  f(): Int { a <- a + d };
}

Class C inherits A {
  c: Int <- 3;
  h(): Int { a <- a * c };
}

Dynamic Dispatch Example

- \( e.g() \)
  - \( g \) refers to method in \( B \) if \( e \) is a \( B \)
- \( e.f() \)
  - \( f \) refers to method in \( A \) if \( f \) is an \( A \) or \( C \)
    (inherited in the case of \( C \))
  - \( f \) refers to method in \( B \) for a \( B \) object

- The implementation of methods and dynamic dispatch strongly resembles the implementation of attributes

Dispatch Tables

- Every class has a fixed set of methods (including inherited methods)
  - A dispatch table indexes these methods
    - An array of method entry points
    - A method \( f \) lives at a fixed offset in the dispatch table for a class and all of its subclasses
Dispatch Table Example

<table>
<thead>
<tr>
<th>Offset Class</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>fA</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>fB</td>
<td>g</td>
</tr>
<tr>
<td>C</td>
<td>fA</td>
<td>h</td>
</tr>
</tbody>
</table>

- The dispatch table for class A has only 1 method
- The tables for B and C extend the table for A to the right
- Because methods can be overridden, the method for f is not the same in every class, but is always at the same offset

Using Dispatch Tables

- The dispatch pointer in an object of class X points to the dispatch table for class X
- Every method f of class X is assigned an offset $O_f$ in the dispatch table at compile time

Using Dispatch Tables (Cont.)

- To implement a dynamic dispatch $e.f()$ we
  - Evaluate e, giving an object x
  - Call $D[O_f]$
    - $D$ is the dispatch table for x
    - In the call, self is bound to x