Code Generation

Lecture 19
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 (http://pages.cs.wisc.edu/~larus/spim.html)
A Sample of MIPS Instructions

- \texttt{lw reg}_1 \texttt{ offset(reg}_2\texttt{)}
  \begin{itemize}
  \item Load 32-bit word from address \texttt{reg}_2 + \texttt{offset} into \texttt{reg}_1
  \end{itemize}

- \texttt{add reg}_1 \texttt{ reg}_2 \texttt{ reg}_3
  \begin{itemize}
  \item \texttt{reg}_1 \leftarrow \texttt{reg}_2 + \texttt{reg}_3
  \end{itemize}

- \texttt{sw reg}_1 \texttt{ offset(reg}_2\texttt{)}
  \begin{itemize}
  \item Store 32-bit word in \texttt{reg}_1 at address \texttt{reg}_2 + \texttt{offset}
  \end{itemize}

- \texttt{addiu reg}_1 \texttt{ reg}_2 \texttt{ imm}
  \begin{itemize}
  \item \texttt{reg}_1 \leftarrow \texttt{reg}_2 + \texttt{imm}
  \item “u” means overflow is not checked
  \end{itemize}

- \texttt{li reg imm}
  \begin{itemize}
  \item \texttt{reg} \leftarrow \texttt{imm}
  \end{itemize}
MIPS Assembly. Example.

• The stack-machine code for $7 + 5$ in MIPS:

  acc ← 7
  push acc
  acc ← 5
  acc ← acc + top_of_stack
  pop

  li $a0 7
  sw $a0 0($sp)
  addiu $sp $sp -4
  li $a0 5
  lw $T1 4($sp)
  add $a0 $a0 $T1
  addiu $sp $sp 4

• We now generalize this to a simple language...
A Small Language

- A language with integers and integer operations

\[
\begin{align*}
P & \rightarrow D; P | D \\
D & \rightarrow \text{def id(ARGS)} = E; \\
ARGS & \rightarrow \text{id, ARGs} | \text{id} \\
E & \rightarrow \text{int} | \text{id} | \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4 \\
& \quad | E_1 + E_2 | E_1 - E_2 | \text{id}(E_1, \ldots, E_n)
\end{align*}
\]
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:

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

  \[ \text{cgen}(i) = \text{li} \; \$a0 \; i \]

- This preserves the stack, as required

- Color key:
  - **RED**: compile time (i.e., cgen is the compiler codegen)
  - **BLUE**: run time (i.e., generated code)
Code Generation for Add

\[
cgen(e_1 + e_2) = \\
cgen(e_1) \\
sw \$a0 0(\$sp) \\
addiu \$sp \$sp -4 \\
cgen(e_2) \\
lw \$T1 4(\$sp) \\
add \$a0 \$T1 \$a0 \\
addiu \$sp \$sp 4
\]

\[
cgen(e_1 + e_2) = \\
cgen(e_1) \\
print “sw \$a0 0(\$sp)” \\
print “addiu \$sp \$sp -4” \\
cgen(e_2) \\
print “lw \$T1 4(\$sp)” \\
print “add \$a0 \$T1 \$a0” \\
print “addiu \$sp \$sp 4”
\]
Code Generation for Add. Wrong!

- Optimization: Put the result of $e_1$ directly in $T1$?

$$cgen(e_1 + e_2) =
\begin{align*}
  &cgen(e_1) \\
  &move \$T1 \$a0 \\
  &cgen(e_2) \\
  &add \$a0 \$T1 \$a0
\end{align*}$$

- Try to generate code for: $3 + (7 + 5)$
Code Generation Notes

• The code for $+$ is a template with recursive calls for code for evaluating $e_1$ and $e_2$

• Stack machine code generation is recursive

• 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 reg₁ reg₂ reg₃`
  - Implements \( reg₁ ← reg₂ - reg₃ \)
  
  \[
cgen(e₁ - e₂) = \]
  
  \[
  \begin{align*}
  &\text{cgen}(e₁) \\
  &\text{sw } a₀ \ 0($sp) \\
  &\text{addiu } sp \ sp -4 \\
  &\text{cgen}(e₂) \\
  &\text{lw } T₁ \ 4($sp) \\
  &\text{sub } a₀ \ T₁ \ a₀ \\
  &\text{addiu } sp \ sp \ 4
  \end{align*}
  \]
Code Generation for Conditional

• We need flow control instructions

• New instruction: `beq reg₁ reg₂ label`
  - Branch to label if $reg₁ = reg₂$

• New instruction: `b label`
  - Unconditional jump to label
cgen(if $e_1 = e_2$ then $e_3$ else $e_4$) =
  cgen($e_1$)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen($e_2$)
  lw $T1 4($sp)
  addiu $sp $sp 4
  beq $a0 $T1 true_branch
false_branch:
  cgen($e_4$)
  b end_if
true_branch:
  cgen($e_3$)
end_if:
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 \( f(x_1, \ldots, x_n) \) 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

• 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:

```
   FP
   ┌─ old fp ─┐
   │      │
   │ x     │
   │      │
   └──────┘
   SP
```

AR of f
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)) =
sw $fp 0($sp)
addiu $sp $sp -4
cgen(e_n)
sw $a0 0($sp)
addiu $sp $sp -4
...
cgen(e_1)
sw $a0 0($sp)
addiu $sp $sp -4
jal f_entry

- 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: \texttt{jr reg}
  - Jump to address in register \texttt{reg}

\[\text{cgen(def f(x_1,\ldots,x_n) = e)} =\]
\[
\begin{align*}
\text{move } &\$fp \$sp \\
\text{sw } &\$ra 0(\$sp) \\
\text{addiu } &\$sp \$sp -4 \\
\text{cgen(e)} \\
\text{lw } &\$ra 4(\$sp) \\
\text{addiu } &\$sp \$sp z \\
\text{lw } &\$fp 0(\$sp) \\
\text{jr } &\$ra
\end{align*}
\]

- 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)**

<table>
<thead>
<tr>
<th>Before call</th>
<th>On entry</th>
<th>Before exit</th>
<th>After call</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>FP</td>
<td>FP</td>
<td>FP</td>
</tr>
<tr>
<td>SP</td>
<td>old fp</td>
<td>old fp</td>
<td>SP</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Prof. Aiken  Lecture 12 (Modified by Prof. Vijay Ganesh)
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.)

- Solution: use a frame pointer
  - Always points to the return address on the stack
  - Since it does not move it can be used to find the variables

- Let $x_i$ be the $i^{th}$ ($i = 1, \ldots, n$) formal parameter of the function for which code is being generated

$$c\text{gen}(x_i) = lw \; $a0 \; z($fp) \quad (z = 4*i)$$
Code Generation for Variables (Cont.)

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

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

• \( X \) is at \( fp + 4 \)
• \( Y \) 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

• It is easy to write a code-generator for a stack machine
Summary

• Production compilers do different things
  - Emphasis is on keeping values (esp. current stack frame) in registers
  - Intermediate results are laid out in the AR, not pushed and popped from the stack
Code Generation for OO Languages

Topic II
Object Layout

• **OO implementation = Stuff from last part + more stuff**

• **OO Slogan:** If \( B \) is a subclass of \( A \), than 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 many languages 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
Typical Object Layout

- The first 3 words of objects contain header information:

<table>
<thead>
<tr>
<th>Class Tag</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Size</td>
<td>4</td>
</tr>
<tr>
<td>Dispatch Ptr</td>
<td>8</td>
</tr>
<tr>
<td>Attribute 1</td>
<td>12</td>
</tr>
<tr>
<td>Attribute 2</td>
<td>16</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Typical Object Layout (Cont.)

- Class tag is an integer
  - Identifies class of the object
- Object size is an integer
  - Size of the object in words
- Dispatch ptr is a pointer to a table of methods
  - More later
- Attributes in subsequent slots
- Lay out in contiguous memory
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)
## Layout Picture

<table>
<thead>
<tr>
<th>Offset 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><strong>A</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Atag</td>
<td>5</td>
<td>*</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td><strong>B</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Btag</td>
<td>6</td>
<td>*</td>
<td>a</td>
<td>d</td>
<td>b</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Ctag</td>
<td>6</td>
<td>*</td>
<td>a</td>
<td>d</td>
<td>c</td>
</tr>
</tbody>
</table>
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>Header</th>
<th>$A_1$ object</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$ attrs.</td>
<td>$A_2$ object</td>
</tr>
<tr>
<td>$A_2$ attrs</td>
<td>$A_3$ object</td>
</tr>
<tr>
<td>$A_3$ attrs</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 };  
    g(): Int { a <- a - b };
};

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