Code Generation

Lecture 12

Instructor: Fredrik Kjolstad
Slide design by Prof. Alex Aiken, with modifications
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<sub>1</sub> offset(reg<sub>2</sub>)**
  - Load 32-bit word from address \( \text{reg}_2 + \text{offset} \) into \( \text{reg}_1 \)
- **add reg<sub>1</sub> reg<sub>2</sub> reg<sub>3</sub>**
  - \( \text{reg}_1 \leftarrow \text{reg}_2 + \text{reg}_3 \)
- **sw reg<sub>1</sub> offset(reg<sub>2</sub>)**
  - Store 32-bit word in \( \text{reg}_1 \) at address \( \text{reg}_2 + \text{offset} \)
- **addiu reg<sub>1</sub> reg<sub>2</sub> imm**
  - \( \text{reg}_1 \leftarrow \text{reg}_2 + \text{imm} \)
  - “u” means overflow is not checked
- **li reg imm**
  - \( \text{reg} \leftarrow \text{imm} \)
MIPS Assembly. Example.

- The stack-machine code for \(7 + 5\) in MIPS:

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

  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*

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

\[
def \text{fib}(x) = \begin{cases} 
0 & \text{if } x = 1 \\
1 & \text{if } x = 2 \\
\text{fib}(x - 1) + \text{fib}(x - 2) & \text{otherwise}
\end{cases}
\]
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:
  \[ \text{cgen}(i) = \text{li} \ $a0 \ i \]

• This preserves the stack, as required

• Color key:
  - RED: compile time
  - BLUE: run time
Code Generation for Add

cgen(e₁ + e₂) =
cgen(e₁)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e₂)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4

cgen(e₁ + e₂) =
cgen(e₁)
print “sw $a0 0($sp)”
print “addiu $sp $sp -4”
cgen(e₂)
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 $t1$?

\[
cgen(e_1 + e_2) = \\
cgen(e_1) \\
move \ t1 \ a0 \\
cgen(e_2) \\
add \ a0 \ t1 \ a0
\]

- 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 reg1 reg2 reg3`
  - Implements `reg1 ← reg2 - reg3`

\[
cgen(e_1 - e_2) = \\
cgen(e_1) \\
sw \$a0 0(\$sp) \\
addiu \$sp \$sp -4 \\
cgen(e_2) \\
lw \$t1 4(\$sp) \\
sub \$a0 \$t1 \$a0 \\
addiu \$sp \$sp 4
\]
Code Generation for Conditional

• We need flow control instructions

• New instruction: \texttt{beq reg}_1 \texttt{ reg}_2 \texttt{ label}
  - Branch to label if \( \texttt{reg}_1 = \texttt{reg}_2 \)

• New instruction: \texttt{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
true_branch:
cgen($e_3$)
end_if:
false_branch:
cgen($e_4$)
b 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
  - 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:

```
FP
   old fp
   y
   x
SP
```

\( \{ \text{AR of } f \} \)
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(e₁,...,eₙ)) =
sw $fp 0($sp)
addiu $sp $sp -4
cgen(eₙ)
sw $a0 0($sp)
addiu $sp $sp -4
...
cgen(e₁)
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}

\begin{verbatim}
cgen(def f(x_1,...,x_n) = e) =
move $fp $sp
sw $ra 0($sp)
addiu $sp $sp -4
cgen(e)
lw $ra 4($sp)
addiu $sp $sp z
lw $fp 0($sp)
jr $ra
\end{verbatim}

- 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>SP</td>
<td>FP</td>
</tr>
<tr>
<td>SP</td>
<td>old fp</td>
<td>y</td>
<td>return</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Before call:**
- FP
- SP

**On entry:**
- FP
- old fp
- y
- x

**Before exit:**
- old fp
- y
- x

**After call:**
- FP
- SP
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 $i^{th}$ ($i = 1, ..., n$) formal parameter of the function for which code is being generated

\[
cgen(x_i) = lw \ a0 \ z($fp) \quad (z = 4^*i)
\]
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:

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

• \( X \) 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

• We recommend you use a stack machine for your Cool compiler (it’s simple)
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
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 then 0 else
    if x = 2 then 1 else
    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 $\text{NT}(e) = \# \text{ of temps needed to evaluate } e$

• $\text{NT}(e_1 + e_2)$
  - Needs at least as many temporaries as $\text{NT}(e_1)$
  - Needs at least as many temporaries as $\text{NT}(e_2) + 1$

• Space used for temporaries in $e_1$ can be reused for temporaries in $e_2$
The Equations

\[ NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2)) \]
\[ NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2)) \]
\[ NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4)) \]
\[ NT(\text{id}(e_1, \ldots, e_n)) = \max(NT(e_1), \ldots, NT(e_n)) \]
\[ NT(\text{int}) = 0 \]
\[ NT(\text{id}) = 0 \]

Is this bottom-up or top-down?
What is \( NT(\ldots\text{code for fib}\ldots) \)?
The Revised AR

• For a function definition $f(x_1,\ldots,x_n) = e$ the AR has $2 + n + NT(e)$ elements
  - Return address
  - Frame pointer
  - $n$ arguments
  - $NT(e)$ locations for intermediate results
<table>
<thead>
<tr>
<th>Old FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_n$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>$x_1$</td>
</tr>
<tr>
<td>Return Addr.</td>
</tr>
<tr>
<td>Temp NT(e)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Temp 1</td>
</tr>
</tbody>
</table>
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(e₁ + e₂) =

cgen(e₁)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e₂)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
Code Generation for + (revised)

cgen(e₁ + e₂, nt) =
  cgen(e₁, nt)
  sw $a0 nt($fp)
  cgen(e₂, nt + 4)
  lw $t1 nt($fp)
  add $a0 $t1 $a0
Notes

• The temporary area is used like a small, fixed-size stack

• Exercise: Write out *cgen* for other constructs
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 $A$, 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>Class Tag</th>
<th>Object Size</th>
<th>Dispatch Ptr</th>
<th>Attribute 1</th>
<th>Attribute 2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Offset
Cool 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>A</td>
<td>Atag</td>
<td>5</td>
<td>*</td>
<td>a</td>
<td>d</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>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 $A_n < ... < A_3 < A_2 < A_1$

<table>
<thead>
<tr>
<th>Header</th>
<th>$A_1$ object</th>
<th>What about multiple inheritance?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$ attrs.</td>
<td>$A_2$ object</td>
<td></td>
</tr>
<tr>
<td>$A_2$ attrs</td>
<td>$A_3$ object</td>
<td></td>
</tr>
<tr>
<td>$A_3$ attrs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</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 };
    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 Class</th>
<th>0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>fA</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 \texttt{e.f()} we
  - Evaluate \texttt{e}, giving an object \texttt{x}
  - Call \texttt{D}[\texttt{O}_f]
    • \texttt{D} is the dispatch table for \texttt{x}
    • In the call, \texttt{self} is bound to \texttt{x}