Code Generation I

- Stack machines
- The MIPS assembly language
- A simple source language
- Stack-machine implementation of the simple language
- Readings: 9.1-9.7

Stack Machines

- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results
- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack
  - Computes the required operation on them
  - Pushes the result on the stack
Example of Stack Machine Operation

- The addition operation on a stack machine

```
5
7
9
...
```
```
7
9
...
```
```
+  
```
```
12
9
...
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Why Use a Stack Machine?

- Each operation takes operands from the same place and puts results in the same place

- This means a uniform compilation scheme

- And therefore a simpler compiler

Why Use a Stack Machine?

- Location of the operands is implicit
  - Always on the top of the stack

- No need to specify operands explicitly

- No need to specify the location of the result

- Instruction “add” as opposed to “add r1, r2”
  ⇒ Smaller encoding of instructions
  ⇒ More compact programs

- This is one reason why Java Bytecodes use a stack evaluation model
Optimizing the Stack Machine

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
- Idea: keep the top of the stack in a register (called accumulator)
  - Register accesses are faster
- The “add” instruction is now
  acc ← acc + top_of_stack
  - Only one memory operation!

Stack Machine with Accumulator

Invariants

- The result of computing an expression is always in the accumulator
- For an operation $\text{op}(e_1,\ldots,e_n)$ push the accumulator on the stack after computing each of $e_1,\ldots,e_{n-1}$
  - After the operation pop $n-1$ values
- After computing an expression the stack is as before
Stack Machine with Accumulator.

Example

Compute $7 + 5$ using an accumulator

A Bigger Example: $3 + (7 + 5)$

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>&lt;init&gt;</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>3, &lt;init&gt;</td>
</tr>
<tr>
<td>acc ← 7</td>
<td>7</td>
<td>3, &lt;init&gt;</td>
</tr>
<tr>
<td>push acc</td>
<td>7</td>
<td>7, 3, &lt;init&gt;</td>
</tr>
<tr>
<td>acc ← 5</td>
<td>5</td>
<td>7, 3, &lt;init&gt;</td>
</tr>
<tr>
<td>acc ← acc + top_of_stack</td>
<td>12</td>
<td>7, 3, &lt;init&gt;</td>
</tr>
<tr>
<td>pop</td>
<td>12</td>
<td>3, &lt;init&gt;</td>
</tr>
<tr>
<td>acc ← acc + top_of_stack</td>
<td>15</td>
<td>3, &lt;init&gt;</td>
</tr>
<tr>
<td>pop</td>
<td>15</td>
<td>&lt;init&gt;</td>
</tr>
</tbody>
</table>
Notes
- It is very important that the stack is preserved across the evaluation of a subexpression
  - Stack before the evaluation of $7 + 5$ is $3$, $<\text{init}>$
  - Stack after the evaluation of $7 + 5$ is $3$, $<\text{init}>$
  - The first operand is on top of the stack

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)
A Sample of MIPS Instructions

- lw reg₁ offset(reg₂)
  - Load 32-bit word from address reg₂ + offset into reg₁
- add reg₁ reg₂ reg₃
  - reg₁ ← reg₂ + reg₃
- sw reg₁ offset(reg₂)
  - Store 32-bit word in reg₁ at address reg₂ + offset
- addiu reg₁ reg₂ imm
  - reg₁ ← reg₂ + 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

  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

\[ P \rightarrow D; \ P \mid D \]
\[ D \rightarrow \text{def id(ARGS) = E;} \]
\[ \text{ARGS} \rightarrow \text{id, ARGSS} \mid \text{id} \]
\[ E \rightarrow \text{int} \mid \text{id} \mid \text{if E}_1 = \text{E}_2 \text{ then E}_3 \text{ else E}_4 \]
\[ \mid \text{E}_1 + \text{E}_2 \mid \text{E}_1 - \text{E}_2 \mid \text{id(\text{E}_1, \ldots, \text{E}_n)} \]

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:

  \[ cgen(i) = li \, a0, i \]

- Note that this also preserves the stack, as required
Code Generation for Add

\[
c\text{gen}(e_1 + e_2) = \\
c\text{gen}(e_1) \\
\text{sw } $a0 0($sp) \\
\text{addiu } $sp $sp -4 \\
c\text{gen}(e_2) \\
\text{lw } $t1 4($sp) \\
\text{add } $a0 $t1 $a0 \\
\text{addiu } $sp $sp 4
\]

- Possible optimization: Put the result of \(e_1\) directly in register \($t1\)?

Code Generation for Add. Wrong!

- Optimization: Put the result of \(e_1\) directly in \($t1\)?

\[
c\text{gen}(e_1 + e_2) = \\
c\text{gen}(e_1) \\
\text{move } $t1 $a0 \\
c\text{gen}(e_2) \\
\text{add } $a0 $t1 $a0
\]

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

- The code for + is a template with "holes" for code for evaluating e₁ and e₂
- Stack machine code generation is recursive
- Code for e₁ + e₂ consists of code for e₁ and e₂ 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(e₁ - e₂) =
  
  cgen(e₁)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen(e₂)
  lw $t1 4($sp)
  sub $a0 $t1 $a0
  addiu $sp $sp 4
Code Generation for Conditional

- We need flow control instructions

- New instruction: `beq reg\_1 reg\_2 label`
  - Branch to label if `reg\_1 = reg\_2`

- New instruction: `b label`
  - Unconditional jump to label

---

Code Generation for If (Cont.)

cgen(if e\_1 = e\_2 then e\_3 else e\_4) =

true\_branch:
cgen(e\_3)
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 activation record
- 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
- It’s handy to have a pointer to the current activation
  - 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 will be:

```
<table>
<thead>
<tr>
<th>FP</th>
<th>old fp</th>
<th>y</th>
<th>x</th>
<th>AR of f</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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} \texttt{reg}
  ■ Jump to address in register \texttt{reg}

cgen(def f(x₁,...,xₙ) = 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

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

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
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

\[
\text{cgen}(x_i) = \text{lw} \ \$a0 \ \text{z}($fp) \quad (z = 4*i)
\]

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

<table>
<thead>
<tr>
<th>FP</th>
<th>( \text{old fp} )</th>
<th>( y )</th>
<th>( x )</th>
<th>return</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>( \text{X is at fp + 4} )</td>
<td>( \text{Y is at fp + 8} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- 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
- Next time: code generation for objects