Code Generation

Lecture 20

Dr. Sean Peisert – ECS 142 – Spring 2009
Status

- 1.5 weeks to go on project 3
- Read Sec. 7.1 through 7.4 by next Monday, May 18
Stack Machines

• A simple evaluation model
• No variables or registers
• A stack of values for intermediate results
Example of a Stack Machine Program

- Consider two instructions
  - push i — place the int i on top of the stack
  - add — pop two elements, add them, and put the result back on the stack

- A program to compute 7 + 5:
  - push 7
  - push 5
  - add
Stack Machine. Example.

- Each instruction
  - takes 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
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:
  \[
  \text{acc} \leftarrow \text{acc} + \text{top\_of\_stack}
  \]
• 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 $e_1, \ldots, e_{n-1}$
  - The result of $e_n$ is in the accumulator before $\text{op}$
  - After the operation, pop $n-1$ values
  - After computing an expression the stack is as before
Stack Machine w/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, too.

• Stack before evaluation of 7+5 is 3, <init>

• Stack after the evaluation of 7+5 is 3, <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 implement 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 toward 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.
Stack

- The stack grows toward lower addresses
- 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 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)
- We’ll give you a SPIM document for more details.
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.

- Stack machine code for 7+5 in MIPS:
  
  \[
  \begin{align*}
  &\text{acc }\leftarrow 7 &\text{li }$a0 7 \\
  &\text{push acc} &\text{sw }$a0 0($sp) \\
  &\text{acc }\leftarrow 5 &\text{addiu }$sp $sp -4 \\
  &\text{acc }\leftarrow \text{acc + top_of_stack} &\text{li }$a0 5 \\
  &\text{pop} &\text{lw }$t1 4($sp) \\
  & &\text{add }$a0 $a0 $t1 \\
  & &\text{addiu }$sp $sp 4
  \end{align*}
  \]

- We now generalize this to a simple language:
A Small Language

- A language with integers and integer operations

\[
P \rightarrow D; P | D
\]
\[
D \rightarrow \text{def } \text{id}(\text{ARGS}) = E;
\]
\[
\text{ARGS} \rightarrow \text{id}, \text{ARGS} | \text{id}
\]
\[
E \rightarrow \text{int} | \text{id} | \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4
\]
\[
| E_1 + E_2 | E_1 - E_2 | \text{id}(E_1, \ldots, E_n)
\]
A Small Language

• 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:

\[
\text{def fib}(x) = \begin{cases} 
0 & \text{if } x = 1 \\
1 & \text{if } x = 2 \\
\text{fib}(x-1) + \text{fib}(x-2) & \text{else}
\end{cases}
\]
• 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:

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

- Note that this also preserves the stack, as required.
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

• Possible optimization: Put the result of $e₁$ directly in register $t₁$?
cgen(e₁ + e₂) =
  cgen(e₁)
  move $t₁ $a0
  cgen(e₂)
  add $a0 $t₁ $a0

• Possible optimization: Put the result of e₁ directly in
  register $t₁ ?
• No -- try to generate code for 3 + (7 + 5)
• 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 $e_1 + e_2$ consists of 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 & Consts

- New instruction: sub reg\_1 reg\_2 reg\_3
- Implements reg\_1 ← reg\_2 - reg\_3

\[
cgen(e\_1 - e\_2) =
\]
\[
cgen(e\_1) \\
sw \$a0 0(\$sp) \\
addiu \$sp \$sp - 4 \\
cgen(e\_2) \\
lw \$t1 4(\$sp) \\
sub \$a0 \$t1 \$a0 \\
addiu \$sp \$sp 4
\]
Code Generation for Conditional

• We need flow control instructions

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

• New instruction: b label
  • Unconditional jump to label
Code Generation for If

cgen(if e₁ = e₂ then e₃ else e₄) =
cgen(e₁)
sw $a0 0($sp)
addiu $sp $sp - 4
cgen(e₂)
lw $t₁ 4($sp)
addiu $sp $sp 4
beq $a0 $t₁ true_branch
false_branch:
cgen(e₄)
b end_if
true_branch:
cgen(e₃)
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

• The stack discipline guarantees that on function exit $sp is the same as it was on function entry

• No need to save $sp

• We need the return address

• It’s handy to have a pointer to start of 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>AR of f</th>
</tr>
</thead>
<tbody>
<tr>
<td>old FP</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
```

High Addresses