Code Generation

Lecture 29
(based on slides by R. Bodik)

Lecture Outline
- Stack machines
- The MIPS assembly language
- The x86 assembly language
- A simple source language
- Stack-machine implementation of the simple language

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 integer 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 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

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 the Java Virtual Machine uses 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 (at least) the top of the stack in a register (called accumulator)
  - Register accesses are faster
- The "add" instruction is now
  - \( \text{acc} ← \text{acc} + \text{top} \)
  - Only one memory operation

Stack Machine with Accumulator

Invariants
- The result of computing an expression is always in the accumulator
- For an operation \( \text{op}(e₁, ..., eₙ) \) push the accumulator on the stack after computing each of \( e₁, ..., eₙ \)
  - The result of \( e₁ \) is in the accumulator before \( \text{op} \)
  - After the operation \( \text{pop} \ n-1 \) values
- After computing an expression the stack is as before

Stack Machine with Accumulator. Example

- Compute \( 7 \times 5 \) using an accumulator

<table>
<thead>
<tr>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

A Bigger Example: \( 3 \times (7 + 5) \)

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>( \langle \text{init} \rangle )</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>3, ( \langle \text{init} \rangle )</td>
</tr>
<tr>
<td>acc ← 7</td>
<td>7</td>
<td>( \langle \text{init} \rangle )</td>
</tr>
<tr>
<td>push acc</td>
<td>7</td>
<td>7, ( \langle \text{init} \rangle )</td>
</tr>
<tr>
<td>acc ← 5</td>
<td>5</td>
<td>7, ( \langle \text{init} \rangle )</td>
</tr>
<tr>
<td>acc ← acc + top_of_stack</td>
<td>12</td>
<td>7, ( \langle \text{init} \rangle )</td>
</tr>
<tr>
<td>pop</td>
<td>12</td>
<td>3, ( \langle \text{init} \rangle )</td>
</tr>
<tr>
<td>acc ← acc + top_of_stack</td>
<td>15</td>
<td>3, ( \langle \text{init} \rangle )</td>
</tr>
<tr>
<td>pop</td>
<td>15</td>
<td>( \langle \text{init} \rangle )</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 \( 3, \langle \text{init} \rangle \)
  - Stack after the evaluation of \( 7 + 5 \) is \( 3, \langle \text{init} \rangle \)
  - 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 an x86 or MIPS processor (or simulator)
- We implement stack machine instructions using MIPS instructions and registers

MIPS assembly vs. x86 assembly

- In Project 3, you will generate x86 code
  - because we have no MIPS machines around
  - and using a MIPS simulator is less exciting
- In this lecture, we will use MIPS assembly
  - it's somewhat more readable than x86 assembly
  - e.g. in x86, both store and load are called mov
  - translation from MIPS to x86 trivial for the restricted subset we'll need
  - see the translation table in a few slides

Simulating a Stack Machine...

- The accumulator is kept in MIPS register $a0
  - in x86, it's in %eax
- The stack is kept in memory
- The stack grows towards lower addresses
  - standard convention on both MIPS and x86
- 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
  - in x86, it's %esp

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 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
- hi reg, imm
  - reg := imm

x86 Assembly

x86 architecture
- Complex Instruction Set Computer (CISC) architecture
- Arithmetic operations can use both registers and memory for operands and results
  - So, you don't have to use separate load and store instructions to operate on values in memory
  - CISC gives us more freedom in selecting instructions (hence, more powerful optimizations)
- but we'll use a simple RISC subset of x86
  - so translation from MIPS to x86 will be easy
**x86 assembly**

- x86 has two-operand instructions:
  - ex.: ADD dest, src  
  - in MIPS: dest := src1 + src2

- An annoying fact to remember ☹️
  - different x86 assembly versions exist
  - one important difference: order of operands
  - the manuals assume
    - ADD dest, src
  - the gcc assembler we’ll use uses opposite order
    - ADD src, dest

**Sample x86 instructions (gcc order of operands)**

- movl offset(reg2), reg0
- add reg2, reg0
- movl offset(reg1)
- store 32-bit word in reg1 at address reg2 + offset
- add imm, reg2
- reg := imm

**MIPS to x86 translation**

<table>
<thead>
<tr>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw reg1, offset(reg2)</td>
<td>movl offset(reg2), reg0</td>
</tr>
<tr>
<td>add reg2, reg1, reg1</td>
<td>add reg2, reg1</td>
</tr>
<tr>
<td>sw reg0, offset(reg1)</td>
<td>movl reg0, offset(reg1)</td>
</tr>
<tr>
<td>addiu reg1, reg0, imm</td>
<td>add imm, reg1</td>
</tr>
<tr>
<td>li reg, imm</td>
<td>movl imm, reg</td>
</tr>
</tbody>
</table>

**x86 vs. MIPS registers**

<table>
<thead>
<tr>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>%eax</td>
</tr>
<tr>
<td>$sp</td>
<td>%esp</td>
</tr>
<tr>
<td>$fp</td>
<td>%ebp</td>
</tr>
<tr>
<td>$t</td>
<td>%ebx</td>
</tr>
</tbody>
</table>

**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...

**Some Useful Macros**

- We define the following abbreviation
  - push $t
  - sw $t, 0($sp)
  - addiu $sp, $sp, -4

- pop
  - addiu $sp, $sp, 4

- $t ← top
  - lw $t, 4($sp)
Useful Macros, IA32 version (GNU syntax)

- `push %t`  
  ```
  pushl %t  
  (t a general register)
  ```

- `pop`  
  ```
  addl $4, %esp  
  or  
  popl %t (also moves top to %t)
  ```

- `%t ← top`  
  ```
  movl (%esp), %t
  ```

A Small Language

- A language with integers and integer operations

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

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) = \text{if } x = 1 \text{ then } 0 \text{ else}
  \text{if } x = 2 \text{ then } 1 \text{ else}
  \text{fib}(x - 1) + \text{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 \( \text{cgen}(e) \)
  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) = li \$a0, i
  ```

  Note that this also preserves the stack, as required

Code Generation for Add

  ```
  \text{cgen}(e_1 + e_2) =
  \text{cgen}(e_1)
  \text{push } \$a0
  \text{cgen}(e_2)
  \$t1 ← \text{top}
  \text{add } \$a0, \$t1, \$a0
  \text{pop}
  ```

  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 $t1$?
  \[
  \text{cgen}(e_1 + e_2) = \\
  \begin{align*}
  &\text{cgen}(e_1) \\
  &\text{move} \; $11, \; $a0 \\
  &\text{cgen}(e_2) \\
  &\text{add} \; $a0, \; $11, \; $a0
  \end{align*}
  \]

- 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 $e_1 + e_2$ consists of code for $e_1$ and $e_2$ glued together
- Code generation can be written as a (modified) post-order traversal of the AST
  - At least for expressions

Code Generation for Sub and Constants

- New instruction: sub $reg_1$, $reg_2$, $reg_3$
  - Implements $reg_1 \leftarrow reg_2 - reg_3$
  \[
  \begin{align*}
  &\text{cgen}(e_1 \cdot e_2) = \\
  &\text{cgen}(e_1), \\
  &\text{push} \; $a0 \\
  &\text{cgen}(e_2) \\
  &$t1 \leftarrow \text{top} \\
  &\text{sub} \; $a0, \; $t1, \; $a0 \\
  &\text{pop}
  \end{align*}
  \]

Code Generation for Conditional

- We need flow control instructions
  - New instruction: beq $reg_1$, $reg_2$, label
    - Branch to label if $reg_1 = reg_2$
    - x86: cmpl $reg_1$, $reg_2$
      je label
  - New instruction: b label
    - Unconditional jump to label
    - x86: jmp label

Code Generation for If (Cont.)

\[
\begin{align*}
\text{cgen}(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \\
\text{false_branch} = \text{new_label}() \\
\text{true_branch} = \text{new_label}() \\
\text{false_branch} = \text{new_label}() \\
\text{true_branch} = \text{new_label}() \\
\text{false_branch} = \text{new_label}() \\
\text{true_branch} = \text{new_label}() \\
\text{false_branch} = \text{new_label}() \\
\text{true_branch} = \text{new_label}() \\
\text{false_branch} = \text{new_label}() \\
\text{true_branch} = \text{new_label}() \\
\text{false_branch} = \text{new_label}() \\
\text{true_branch} = \text{new_label}() \\
\text{false_branch} = \text{new_label}() \\
\text{true_branch} = \text{new_label}() \\
\text{false_branch} = \text{new_label}() \\
\text{true_branch} = \text{new_label}() \\
\text{false_branch} = \text{new_label}() \\
\text{true_branch} = \text{new_label}() \\
\text{false_branch} = \text{new_label}() \\
\text{true_branch} = \text{new_label}()
\end{align*}
\]