Code Generation

Lecture 30
(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

push 7
push 5
add

[Diagram showing stack operations]
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 $op(e_1,...,e_n)$ push the accumulator on the stack after computing each of $e_1,...,e_{n-1}$
  - The result of $e_n$ is in the accumulator before $op$
  - 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

```
acc

stack
... 7 ...
... ...
... ...

acc ← 7
push acc

acc ← 5
acc ← acc + top_of_stack
pop

acc ← 12
```
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, <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 an x86 or MIPS processor (or simulator)

• We implement stack machine instructions using MIPS instructions and registers
MIPS assembly vs. x86 assembly

• In Project 4, 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 \texttt{movl}
• translation from MIPS to x86 trivial
  - 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

- \texttt{lw reg}_{1} \texttt{ offset(reg}_{2}\texttt{)}
  \quad \cdot \text{Load 32-bit word from address } reg_{2} + \text{offset into } reg_{1}
- \texttt{add reg}_{1}, \texttt{ reg}_{2}, \texttt{ reg}_{3}
  \quad \cdot \text{reg}_{1} \leftarrow \text{reg}_{2} + \text{reg}_{3}
- \texttt{sw reg}_{1}, \texttt{ offset(reg}_{2}\texttt{)}
  \quad \cdot \text{Store 32-bit word in } \text{reg}_{1} \text{ at address } \text{reg}_{2} + \text{offset}
- \texttt{addiu reg}_{1}, \texttt{ reg}_{2}, \texttt{ imm}
  \quad \cdot \text{reg}_{1} \leftarrow \text{reg}_{2} + \text{imm}
  \quad \cdot \text{“u” means overflow is not checked}
- \texttt{li reg}, \texttt{ imm}
  \quad \cdot \text{reg} \leftarrow \text{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 \[\text{dest := dest + src}\]
  - in MIPS: \[\text{dest := src1 + src2}\]
• An annoying fact to remember 😞
  - different x86 assembly versions exists
  - 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(reg_2), reg_1`
  - Load 32-bit word from address `reg_2 + offset` into `reg_1`
- `add reg_2, reg_1`
  - `reg_1 ← reg_1 + reg_2`
- `movl reg_1 offset(reg_2)`
  - Store 32-bit word in `reg_1` at address `reg_2 + offset`
- `add imm, reg_1`
  - `reg_1 ← reg_1 + imm`
  - use this for MIPS' `addiu`
- `movl imm, reg`
  - `reg ← imm`
## MIPS to x86 translation

<table>
<thead>
<tr>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw reg&lt;sub&gt;1&lt;/sub&gt;, offset(reg&lt;sub&gt;2&lt;/sub&gt;)</td>
<td>movl offset(reg&lt;sub&gt;2&lt;/sub&gt;), reg&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>add reg&lt;sub&gt;1&lt;/sub&gt;, reg&lt;sub&gt;1&lt;/sub&gt;, reg&lt;sub&gt;2&lt;/sub&gt;</td>
<td>add reg&lt;sub&gt;2&lt;/sub&gt;, reg&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>sw reg&lt;sub&gt;1&lt;/sub&gt;, offset(reg&lt;sub&gt;2&lt;/sub&gt;)</td>
<td>movl reg&lt;sub&gt;1&lt;/sub&gt;, offset(reg&lt;sub&gt;2&lt;/sub&gt;)</td>
</tr>
<tr>
<td>addiu reg&lt;sub&gt;1&lt;/sub&gt;, reg&lt;sub&gt;1&lt;/sub&gt;, imm</td>
<td>add imm, reg&lt;sub&gt;1&lt;/sub&gt;</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  
  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...
Some Useful Macros

- We define the following abbreviation

- push $t$
  
  \[
  \text{sw } t, 0(\text{sp})
  \]
  
  \[
  \text{addiu } \text{sp}, \text{sp}, -4
  \]

- pop
  
  \[
  \text{addiu } \text{sp}, \text{sp}, 4
  \]

- $t \leftarrow \text{top}$
  
  \[
  \text{lw } t, 4(\text{sp})
  \]
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

\[
P \rightarrow D; P \mid D \\
D \rightarrow \text{def} \ id(\text{ARGS}) = E; \\
\text{ARGS} \rightarrow id, \text{ARGS} \mid id \\
E \rightarrow \text{int} \mid id \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4 \\
\mid E_1 + E_2 \mid E_1 - E_2 \mid id(E_1, \ldots, E_n)
\]
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:
  
  ```python
  def fib(x) = if x = 1 then 0 else
                if x = 2 then 1 else
                  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:

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

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

cgen(e_1 + e_2) =
   cgen(e_1)
push $a0
cgen(e_2)
$t1 ← top
add $a0, $t1, $a0
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 $t1$?

  \[
  \text{cgen}(e_1 + e_2) = \\
  \qquad \text{cgen}(e_1) \\
  \qquad \text{move } t1, a0 \\
  \qquad \text{cgen}(e_2) \\
  \qquad \text{add } a0, t1, a0
  \]

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

• The code for + is a template with “holes” for code for evaluating $e_1$ 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 reg₁ reg₂ reg₃`
  - Implements `reg₁ ← reg₂ - reg₃`
  
  \[
  \text{cgen}(e₁ - e₂) = \\
  \text{cgen}(e₁) \\
  \text{push } $a₀ \\
  \text{cgen}(e₂) \\
  $t₁ ← \text{top} \\
  \text{sub } $a₀, $t₁, $a₀ \\
  \text{pop}
  \]
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
  - x86: \texttt{cmpl reg}_1, \texttt{reg}_2
    \begin{verbatim}
    je label
    \end{verbatim}

• New instruction: \texttt{b label}
  - Unconditional jump to label
  - x86: \texttt{jmp label}
Code Generation for If (Cont.)

cgen(if e₁ = e₂ then e₃ else e₄) =
  false_branch = new_label ()
  true_branch = new_label ()
  end_if = new_label ()
  cgen(e₁)
  push $a0
  cgen(e₂)
  $t1 ← top
  pop
  beq $a0, $t1, true_branch
  cgen(e₄)
  b end_if
  true_branch:
  cgen(e₃)
  end_if:

false_branch: