Code Generation (I)

Lecture 15

Lecture Outline

- Stack machines
- The MIPS 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
- 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

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

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:
  - Only one memory operation

Stack Machine with Accumulator

Invariants
- The result of computing an expression is always in the accumulator
- For an operation op(e1, …, en) push the accumulator on the stack after computing each of e1, …, en-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

```
acc ← 3
push acc
acc ← 7
push acc
acc ← 5
acc ← acc + top_of_stack
pop
acc ← acc + top_of_stack
pop
```

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>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 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 handout 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.

- The stack-machine code for 7 + 5 in MIPS:
  - acc ← 7
  - push acc
  - acc ← 5
  - acc ← acc + top_of_stack
  - pop
  - Read the SPIM handout for more details

A Small Language

- A language with integers and integer operations
  - P → D; P | D
  - D → def id(ARGS) = E;
  - ARGS → id, ARGS | id
  - E → int | id | if E1 = E2 then E3 else E4
  - E1 + E2 | E1 - E2 | id(E1, ..., E4)
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:
  \[
  \text{cgen}(i) = \text{li } a0 i
  \]
  Note that this also preserves the stack, as required.

Code Generation for Add

- The code to evaluate an addition simply copies the operands into registers and adds them:
  \[
  \text{cgen}(e_1 + e_2) = \\
  \text{cgen}(e_1) \\
  \text{sw } a0 0(sp) \\
  \text{addiu } sp sp -4 \\
  \text{cgen}(e_2) \\
  \text{lw } t1 4(sp) \\
  \text{add } a0 t1 a0 \\
  \text{addiu } sp sp 4 \\
  \]
  - Possible optimization: Put the result of e_1 directly in register $t1?

Code Generation 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 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 reg1 reg2 label
  - Branch to label if reg1 = reg2
- New instruction: b label
  - Unconditional jump to label

**Code Generation for If (Cont.)**

```
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(x1,...,xn) push x1,...,xn 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>
</tr>
</thead>
<tbody>
<tr>
<td>old fp</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>x</td>
</tr>
<tr>
<td>SP</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)) =

1. `sw $fp 0($sp)`
2. `addiu $sp $sp -4`
3. `cgen(en)`
4. `sw $a0 0($sp)`
5. `addiu $sp $sp -4`
6. `…`
7. `cgen(e1)`
8. `sw $a0 0($sp)`
9. `addiu $sp $sp -4`
10. `jal f_entry`

The caller saves its value of the frame pointer.

Then it saves the actual parameters in reverse order.

The AR so far is $4n+4$ bytes long.

**Code Generation for Function Definition**

- New instruction: `jr reg`
  - Jump to address in register reg.

cgen(def f(x1,… , xn) = e) =

1. `move $fp $sp`
2. `sw $ra 0($sp)`
3. `addiu $sp $sp -4`
4. `cgen(e)`
5. `lw $ra 4($sp)`
6. `addiu $sp $sp z`
7. `lw $fp 0($sp)`
8. `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.

$z = 4n + 8$

**Calling Sequence. Example for f(x,y).**

Before call | On entry | Before exit | After call
--- | --- | --- | ---
| FP | SP | FP | SP
| x | y | old fp | old fp
| SP | FP | return | 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.

$$cgen(x_i) = lw \text{ } a0 \text{ } z(fp) \quad (z = 4^i)$$
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:

  \[
  \begin{array}{c}
  \text{old fp} \\
  \text{y} \\
  \text{x} \\
  \text{FP} \\
  \text{return} \\
  \text{SP}
  \end{array}
  \begin{array}{c}
  \text{x} \text{ is at } \text{fp} + 4 \\
  \text{y} \text{ is at } \text{fp} + 8
  \end{array}
  \]

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
- See the Web page for a large code generation example
- Next time: code generation for objects