The Main Idea of Today’s Lecture

We can emit stack-machine-style code for expressions via recursion

(We will use MIPS assembly as our target language)

Lecture Outline

• What are stack machines?
• The MIPS assembly language
• A simple source language ("Mini Bar")
• A 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 onto 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 topmost two elements, add them and put the result back onto 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 is “add” as opposed to “add r1, r2” (or “add rd ri1 ri2”)
  ⇒ Smaller encoding of instructions
  ⇒ More compact programs
• This is one of the reasons why Java Bytecode 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 the top of the stack in a dedicated register (called the “accumulator”)
  - Register accesses are faster (why?)
- 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 placed in the accumulator
- For an operation \( \text{op}(e_1, \ldots, e_n) \) compute each \( e_i \) and then push the accumulator (= the result of evaluating \( e_i \)) onto the stack
- 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
5
7

acc ← 5
push acc
7
acc ← acc + top_of_stack
12
pop

12
```

A Bigger Example: 3 + (7 + 5)

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>&lt;init&gt;</td>
<td></td>
</tr>
<tr>
<td>acc ← 3</td>
<td>3</td>
<td>3, &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>7, 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 the MIPS processor (or simulator)
  • We simulate the stack machine instructions using MIPS instructions and registers

Simulating a Stack Machine on the MIPS...
• 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
  - Guess: what does “sp” stand for?
  - 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 store results in memory
• 32 general purpose registers (32 bits each)
  • We will use $sp, $a0 and $t1 (a temporary register)

Read the SPIM documentation for more details
A Sample of MIPS Instructions

- `lw reg1 offset(reg2)`  
  `“load word”`
  - Load 32-bit word from address `reg2 + offset` into `reg1`
- `add reg1 reg2 reg3`  
  `“store word”`
  - Store 32-bit word in `reg1` at address `reg2 + offset`
- `addiu reg1 reg2 imm`  
  `“add immediate”`
  - `reg1 ← reg2 + imm`
  - “u” means overflow is not checked
- `li reg imm`  
  `“load immediate”`
  - `reg ← imm`

MIPS Assembly: Example

- The stack-machine code for `7 + 5` in MIPS:
  ```assembly`
  acc ← 7
  push acc
  acc ← 5
  acc ← acc + top_of_stack
  addiu $sp $sp -4
  li $a0 7
  sw $a0 0($sp)
  addiu $sp $sp -4
  li $a0 5
  lw $t1 4($sp)
  add $a0 $a0 $t1
  li $a0 4
  addiu $sp $sp 4
  ```

- We now generalize this to a simple language...

A Small Language

- A language with only integers and integer operations (“Mini Bar”)
  ```plaintext`
  P → F P | F
  F → id(ARGS) begin E end
  ARG$S → id, ARG$S | id
  E → int | id | if $E_1 = $E_2 then $E_3 else $E_4
  | $E_1 + $E_2 | $E_1 - $E_2 | id(ES)
  ES → E, ES | E
  ```

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:
  ```plaintext`
  fib(x)
  begin
  if x = 1 then 0 else
  if x = 2 then 1 else fib(x - 1) + fib(x - 2)
  end
  ```
**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 \( e \)
  - \( cgen(e) \) will be recursive

**Code Generation for Constants**

- The code to evaluate an integer constant simply copies it into the accumulator:

  \[
  cgen(int) = li \ a0 \ int
  \]

- Note that this also preserves the stack, as required

**Code Generation for Addition**

\[
\begin{align*}
\text{cgen}(e_1 + e_2) &= \text{cgen}(e_1) \quad ; \ a0 \leftarrow \text{value of } e_1 \\
&\text{sw } a0 \ 0(sp) \quad ; \ \text{push that value} \\
&\text{addiu } sp \ sp -4 \quad \text{onto the stack} \\
&\text{cgen}(e_2) \quad ; \ a0 \leftarrow \text{value of } e_2 \\
&\text{lw } t1 \ 4(sp) \quad ; \ \text{grab value of } e_1 \\
&\text{add } a0 \ t1 \ a0 \quad ; \ \text{do the addition} \\
&\text{addiu } sp \ sp 4 \quad ; \ \text{pop the stack}
\end{align*}
\]

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

**Code Generation for Addition: Wrong Attempt!**

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

\[
\begin{align*}
\text{cgen}(e_1 + e_2) &= \text{cgen}(e_1) \quad ; \ a0 \leftarrow \text{value of } e_1 \\
&\text{move } t1 \ a0 \quad ; \ \text{save that value in } t1 \\
&\text{cgen}(e_2) \quad ; \ a0 \leftarrow \text{value of } e_2 \\
&\text{add } a0 \ t1 \ a0 \quad ; \ \text{may clobber } t1 \\
&\text{addiu } sp \ sp 4 \quad ; \ \text{perform the addition}
\end{align*}
\]

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

- The code for $e_1 + e_2$ 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 (arithmetic) expressions

Code Generation for Subtraction and Constants

New instruction: sub reg1 reg2 reg3
Implements $\text{reg}_1 \leftarrow \text{reg}_2 \,-\, \text{reg}_3$

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

Code Generation for Conditional

- We need flow control instructions
- New MIPS instruction: beq reg1 reg2 label
  - Branch to label if reg1 = reg2
- New MIPS instruction: j label
  - Unconditional jump to label

Code Generation for If (Cont.)

\[
c\text{gen}(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =
\]
\[
c\text{gen}(e_1); \; \text{sw} \; \$a0 \; 0($sp); \; \text{addiu} \; \$sp \; \$sp \,-4
\]
\[
c\text{gen}(e_2);
\]
\[
\text{lw} \; \$t1 \; 4($sp); \; \text{addiu} \; \$sp \; \$sp \,+4 \; \text{beq} \; \$a0 \; \$t1 \; \text{true_branch}
\]
\[
\text{false_branch:} \;
\]
\[
c\text{gen}(e_4); \; \text{addiu} \; \$sp \; \$sp \,+4 \; \text{beq} \; \$a0 \; \$t1 \; \text{true_branch}
\]
\[
\text{end_if:}
\]
Meet The Activation Record

- Code for function calls and function definitions depends on the layout of the activation record (or "AR")
- 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 $f(x_1,...,x_n)$ push the arguments $x_n,...,x_1$ onto the stack
    - These are the only variables in this language

Meet The Activation Record (Cont.)

- The stack discipline guarantees that on function exit, $\text{sp}$ is the same as it was before the args got pushed (i.e., before function call)
- We need the return address
- It's also handy to have a pointer to the current activation
  - This pointer lives in register $\text{fp}$ (frame pointer)
  - Reason for frame pointer will be clear shortly (at least I hope!)

Layout of 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:

```
old FP
y
x
```

Code Generation for Function Call

- The calling sequence is the sequence of 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 special register $\text{ra}$
  - On other architectures the return address is stored on the stack by the "call" instruction
**Code Generation for Function Call (Cont.)**

\[
cgen(f(e_1, \ldots, e_n)) = \\
\quad \text{sw } $fp 0($sp) \\
\quad \text{addiu } $sp $sp -4 \\
\quad \text{cgen}(e_n) \\
\quad \text{sw } $a0 0($sp) \\
\quad \text{addiu } $sp $sp -4 \\
\quad \ldots \\
\quad \text{cgen}(e_1) \\
\quad \text{sw } $a0 0($sp) \\
\quad \text{addiu } $sp $sp -4 \\
\quad \text{jal } f\_\text{entry}
\]

- The caller saves the value of the frame pointer
- Then it pushes the actual parameters in reverse order
- The caller’s jal puts the return address in register $ra
- The AR so far is \(4n+4\) bytes long

**Code Generation for Function Definition**

- **New MIPS instruction:** jr \(\text{reg}\)
  - Jump to address in register \(\text{reg}\)

\[
cgen(f(x_1, \ldots, x_n) \begin{array}{c} \text{begin e} \\ \text{end} \end{array}) = \\
\quad \text{f\_entry:} \\
\quad \text{move } $fp $sp \\
\quad \text{sw } $ra 0($sp) \\
\quad \text{addiu } $sp $sp -4 \\
\quad \text{cgen}(e) \\
\quad \text{lw } $ra 4($sp) \\
\quad \text{addiu } $sp $sp \text{ frame\_size} \\
\quad \text{lw } $fp 0($sp) \\
\quad \text{jr } $ra
\]

- Note: The frame pointer points to the top, not bottom of the frame
- Callee saves old return address, evaluates its body, pops the return address, pops the arguments, and then restores $fp
- frame\_size = \(4n + 8\)

**Calling Sequence: Example for \(f(x,y)\)**

<table>
<thead>
<tr>
<th>Before call</th>
<th>On entry</th>
<th>After body</th>
<th>After call</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP1</td>
<td>FP1</td>
<td>FP1</td>
<td>FP1</td>
</tr>
<tr>
<td>SP</td>
<td>FP1</td>
<td>FP1</td>
<td>SP</td>
</tr>
<tr>
<td>y</td>
<td>x</td>
<td>y</td>
<td>RA</td>
</tr>
</tbody>
</table>

**Code Generation for Variables/Parameters**

- 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/Parameters**

- Solution: use the 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
  
  $\text{cgen}(x_i) = \text{lwa} \ offset(fp) \ ( offset = 4*i )$

**Activation Record & Code Generation Summary**

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

**Code Generation for Variables/Parameters**

- Example: For a function $f(x,y)$ begin e end the activation and frame pointer are set up as follows (when evaluating e):

| old FP | x is at $fp + 4$
|--------|------------------|
| y      | y is at $fp + 8$
| x      |                  |
| RA     |                  |

**Discussion**

- 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
  - As a result, code generation is often performed in synergy with register allocation

**Next time:** code generation for temporaries and a deeper look into parameter passing mechanisms