**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
  - Location of the operands and result implicit
  - Always on the top of the stack
- This means a uniform compilation scheme and therefore a simpler compiler
  - Example: 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 \( e_1, \ldots, e_{n-1} \)
  - 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

\[
\begin{array}{c|c|c}
\text{acc} & \text{stack} \\
\hline
7 & 5 & 12
\end{array}
\]

- \( \text{acc} \leftarrow 7 \)
- \( \text{push acc} \)
- \( \text{acc} \leftarrow 7 \)
- \( \text{pop} \)
- \( \text{acc} \leftarrow 7 + 5 \)
- \( \text{push acc} \)
- \( \text{acc} \leftarrow 7, 3, \text{<init>} \)
- \( \text{acc} \leftarrow 5 \)
- \( \text{push acc} \)
- \( \text{acc} \leftarrow 12 \)
- \( \text{pop} \)
- \( \text{acc} \leftarrow 15 \)

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Acc</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc \leftarrow 3</td>
<td>3</td>
<td>\text{&lt;init&gt;}</td>
</tr>
<tr>
<td>push acc</td>
<td>3</td>
<td>3, \text{&lt;init&gt;}</td>
</tr>
<tr>
<td>acc \leftarrow 7</td>
<td>7</td>
<td>3, \text{&lt;init&gt;}</td>
</tr>
<tr>
<td>push acc</td>
<td>7</td>
<td>7, 3, \text{&lt;init&gt;}</td>
</tr>
<tr>
<td>acc \leftarrow 5</td>
<td>5</td>
<td>7, 3, \text{&lt;init&gt;}</td>
</tr>
<tr>
<td>acc \leftarrow acc + top of stack</td>
<td>12</td>
<td>7, 3, \text{&lt;init&gt;}</td>
</tr>
<tr>
<td>pop</td>
<td>12</td>
<td>3, \text{&lt;init&gt;}</td>
</tr>
<tr>
<td>acc \leftarrow acc + top of stack</td>
<td>15</td>
<td>3, \text{&lt;init&gt;}</td>
</tr>
<tr>
<td>pop</td>
<td>15</td>
<td>\text{&lt;init&gt;}</td>
</tr>
</tbody>
</table>

Notes

- It is very important that the stack is preserved across the evaluation of a sub expression
  - Stack before the evaluation of \( 7 + 5 \) is \( 3, \text{<init>} \)
  - Stack after the evaluation of \( 7 + 5 \) is \( 3, \text{<init>} \)
  - 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 reg, offset(reg2)
  - Load 32-bit word from address reg2 + offset into reg
- add reg, reg2 reg3
  - reg ← reg2 + reg3
- sw reg, offset(reg2)
  - Store 32-bit word in reg2 at address reg2 + offset
- addiu reg, reg2 imm
  - reg ← 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

A Small Language

- A language with integers and integer operations

P → D; P | D
D → def id(ARG5) = E;
ARG5 → id, ARG5 | id
E → int | id | if E1 = E2 then E3 else E4
  | E1 + E2 | E1 - E2 | id(E1,...,En)

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:
  - def fib(x) = if x = 1 then 0 else
    if x = 2 then 1 else
    fib(x - 1) + fib(x - 2)
**Code Generation Strategy (Invariant)**

- 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:
  
  $cgen(i) = li \; a0 \; i$

- Note that this also preserves the stack, as required

**Code Generation for Add**

\[
cgen(e_1 + e_2) =
\]

- $cgen(e_1)$
- $sw \; a0 \; 0(\; sp \;)$
- $addiu \; sp \; sp \; -4$
- $cgen(e_2)$
- $lw \; t1 \; 4(\; sp \;)$
- $add \; a0 \; t1 \; a0$
- $addiu \; sp \; sp \; 4$
- 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 $t1$?

\[
cgen(e_1 + e_2) =
\]

- $cgen(e_1)$
- $move \; t1 \; a0$
- $cgen(e_2)$
- $add \; a0 \; t1 \; a0$

- 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 recursive descent of the AST
  - At least for expressions

**Code Generation for Sub and Constants**

- New instruction: $sub \; reg1 \; reg2 \; reg3$
  - Implements $reg1 \leftarrow reg2 - reg3$
\[
cgen(e_1 - e_2) =
\]

- $cgen(e_1)$
- $sw \; a0 \; 0(\; sp \;)$
- $addiu \; sp \; sp \; -4$
- $cgen(e_2)$
- $lw \; t1 \; 4(\; sp \;)$
- $sub \; a0 \; t1 \; a0$
- $addiu \; sp \; sp \; 4$
**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 (Cont.)**

cgen(if \(e_1 = e_2\) then \(e_3\) else \(e_4\)) =

cgen(\(e_1\))

sw $a0 0($sp)

addiu $sp $sp -4

cgen(\(e_2\))

lw $t1 4($sp)

addiu $sp $sp 4

beq $a0 $t1 true_branch

false_branch:

cgen(\(e_4\))
b end_if

true_branch:

cgen(\(e_3\))

dif 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 \(\text{f}(x_1,\ldots,x_n)\), push \(x_n,\ldots,x_1\) on the stack
    - These are the only variables in this language

**The Activation Record (Cont.)**

- The stack discipline guarantees that on function exit \(\text{sp}\) 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 \(\text{fp}\) (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 \(\text{f}(x,y)\), The AR will be:

```
FP
old fp
y
x
SP
```

**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 \(\text{ra}\)
  - On other architectures, the return address is stored on the stack by the "call" instruction
Code Generation for Function Call (Cont.)

cgen(f(e₁,…,eₙ)) =
sw $fp 0($sp)
addiu $sp $sp -4
cgen(e₁)
sw $a0 0($sp)
addiu $sp $sp -4
....
cgen(eₙ)
sw $a0 0($sp)
addiu $sp $sp -4
jal f_entry

- The caller saves its value of the frame pointer
- Then it saves the actual parameters in reverse order
- The caller saves the return address in register $ra
- 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) =
mov $fp $sp
sw $ra 0($sp)
addiu $sp $sp -4
cgen(e)
lw $ra 4($sp)
addiu $sp $sp 4
lw $fp 0($sp)
jr $ra

- Note: The frame pointer points to the top, not bottom of the frame.
- The callee saves the return address, to enable later calls.
- The callee finally restores the return address, pops the actual arguments, and restores 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 | FP | FP | FP
SP | old FP | old FP | SP

In caller
- SP: x
- FP: y

By caller
- SP: Popped by callee
- FP: return

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 (and later popped by the callee)

• 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 $xᵢ$ be the $i^{th}$ ($i = 1,…,n$) formal parameter of the function for which code is being generated

  cgen($xᵢ$) = lw $a0 z($fp) ($z = 4*ᵢ$)

Code Generation for Variables (Cont.)

• Example: For a function $def f(x,y) = e$ the activation and frame pointer are set up as follows:

  - X is at fp + 4
  - Y is at fp + 8
Summary

• The activation record must be designed together with the code generator
• Code generation can be done by recursive traversal of the AST
  - Use of a stack machine recommended for Cool compiler (it's simple)
• 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