CODE GENERATION

Baishakhi Ray

Fall 2020

These slides are motivated from Prof. Alex Aiken: Compilers (Stanford)
Stack Machine

- 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

pop  add  push
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
  acc ← acc + top_of_stack
  - Only one memory operation!
Stack Machine with Accumulator

- **Invariants**
  - The result of an expression is in the accumulator
  
  - For op($e_1, \ldots, e_n$) push the accumulator on the stack after computing $e_1, \ldots, e_{n-1}$
    - After the operation pops n-1 values
  
  - Expression evaluation preserves the stack
Stack Machine with Accumulator. Example

- Compute 7 + 5 using an accumulator

1. acc ← 7; push acc
2. acc ← 5
3. acc ← acc + top_of_stack
4. 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</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>

It is very important evaluation of a subexpression preserves the stack
- Stack before the evaluation of 7 + 5 is 3
- Stack after the evaluation of 7 + 5 is 3
- The first operand is on top of the stack
If the current state of the stack is: Acc: 3; Stack: 14, < init >; What is the next line of code to generate for the code fragment \((5 + 9) + 3\)?

Consider the expression \((7 + 5) \times (3 + 2)\). Which of the following are possible stack machine states during the evaluation?
From Stack Machines to MIPS

- The compiler generates code for a stack machine with accumulator
- Let’s run the resulting code on a MIPS like processor.
  - Simulate stack machine instructions using MIPS instructions and registers
- The accumulator is kept in MIPS register $a0
- The stack is kept in memory
  - The stack grows towards lower addresses
- The address of the next location on the stack is kept in MIPS register $sp (stack pointer)
  - 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)
A Sample of MIPS Instructions

- **lw reg1 offset(reg2)**
  - Load 32-bit word from the value of reg2 (which is a memory address), add a fixed value 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
The stack-machine code for 7 + 5 in MIPS:

<table>
<thead>
<tr>
<th>Steps</th>
<th>MIPS Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc = 7</td>
<td>li $a0 7</td>
</tr>
<tr>
<td>push acc</td>
<td>sw $a0 0($sp)</td>
</tr>
<tr>
<td></td>
<td>addiu $sp $sp -4</td>
</tr>
<tr>
<td>acc ← 5</td>
<td>li $a0 5</td>
</tr>
<tr>
<td>acc ← acc + top_of_stack</td>
<td>lw $t1 4($sp)</td>
</tr>
<tr>
<td></td>
<td>add $a0 $a0 $t1</td>
</tr>
<tr>
<td>pop</td>
<td>addiu $sp $sp 4</td>
</tr>
</tbody>
</table>

Let’s generalize this to a simple language
A Small Language

- A language with integers and integer operations

\[
P \rightarrow D; \; P \mid D
\]

\[
D \rightarrow \text{def id(ARGS)} = E;
\]

\[
\text{ARGS} \rightarrow \text{id, ARG} \mid \text{id}
\]

\[
E \rightarrow \text{int} \mid \text{id} \mid \text{if} \; E_1 = E_2 \text{ then } E_3 \text{ else } E_4
\]

\[
\quad \mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1, \ldots, E_n)
\]

- 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:

\[
def \text{fib}(x) = \text{if } x = 1 \text{ then } 0 \text{ else }
\]

\[
\quad \text{if } x = 2 \text{ then } 1 \text{ else }
\]

\[
\quad \text{fib}(x - 1) + \text{fib}(x - 2)
\]
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 $cgen(e)$ whose result is the code generated for $e$

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

  $$cgen(i) = li \ a0 \ i$$

- This preserves the stack, as required

- Color key:
  - RED: compile time
  - BLUE: run time
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`

\[ cgen(e_1 + e_2) = \]
- \( cgen(e_1) \)
  - `print “sw $a0 \ 0($sp)”`
- \( cgen(e_2) \)
  - `print “lw \ $t1 \ 4($sp)”`
  - `print “add \ $a0 \ $t1 \ $a0”`
  - `print “addiu \ $sp \ $sp \ 4”`
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
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 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 $f(x_1,\ldots,x_n)$ 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 $sp is the same as it was on function entry
  - No need for a control link

- We need the return address

- A pointer to the current activation is useful
  - 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 is:

```
<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>
</tbody>
</table>
```

AR of f
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
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 $4$n+4$ bytes long

cgen(f(e_1,\ldots,e_n)) =
sw \ $fp \ 0($sp)
addiu \ $sp \ $sp \ -4
cgen(e_n)
sw \ $a0 \ 0($sp)
addiu \ $sp \ $sp \ -4
...
cgen(e_1)
sw \ $a0 \ 0($sp)
addiu \ $sp \ $sp \ -4
jal \ f\_entry
Code Generation for Function Definition

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

\[
cgen(\text{def } f(x_1, ..., x_n) = e) =
\]

    fEntry:
    \begin{align*}
    & \text{move } $fp \text{ } $sp \\
    & \text{sw } $ra \text{ } 0($sp) \\
    & \text{addiu } $sp \text{ } $sp \text{ } -4 \\
    & \text{cgen(e)} \\
    & \text{lw } $ra \text{ } 4($sp) \\
    & \text{addiu } $sp \text{ } $sp \text{ } z \\
    & \text{lw } $fp \text{ } 0($sp) \\
    & \text{jr } $ra
    \end{align*}

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.

\[z = 4*n + 8\]
Calling Sequence: Example for f(x,y)

Before call

On entry

Before exit

After call

FP

old fp

FP

old fp

return

SP

y

SP

x

FP

y

FP

x

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 $sp$
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 \((i = 1, \ldots, n)\) formal parameter of the function for which code is being generated

\[
cgen(x_i) = \text{lw} \ $a0 \ z($fp) \ (\ z = 4*i \ )
\]
Example: For a function \( \text{def } f(x,y) = e \) the activation and frame pointer are set up as follows:

- X is at \( \text{fp} + 4 \)
- Y is at \( \text{fp} + 8 \)
The activation record must be designed together with the code generator.

Code generation can be done by recursive traversal of the AST.

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