

Code Generation

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)

2

Lecture Outline

- What are stack machines?
- The MIPS assembly language
- A simple source language ("Mini Bar")
- A stack machine implementation of the simple language

3

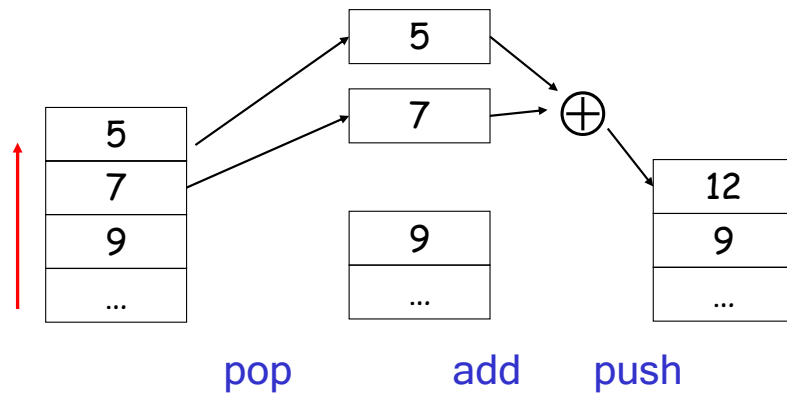
Stack Machines

- 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

4

Example of Stack Machine Operation

The addition operation on a stack machine



5

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
```

6

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

7

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

8

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
 - $acc \leftarrow acc + top_of_stack$
 - Only one memory operation!

9

Stack Machine with Accumulator

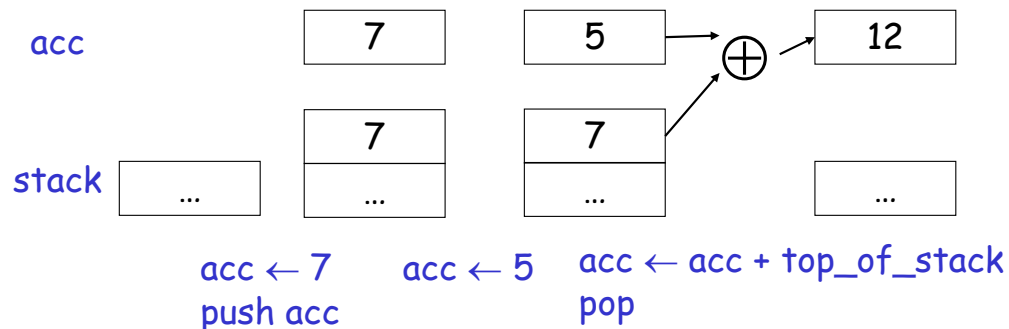
Invariants

- The result of computing an expression is always placed in the accumulator
- For an operation $op(e_1, \dots, e_n)$ 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

10

Stack Machine with Accumulator: Example

Compute $7 + 5$ using an accumulator



11

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

Code	Acc	Stack
$acc \leftarrow 3$	3	<init>
push acc	3	3, <init>
$acc \leftarrow 7$	7	3, <init>
push acc	7	7, 3, <init>
$acc \leftarrow 5$	5	7, 3, <init>
$acc \leftarrow acc + top_of_stack$	12	7, 3, <init>
pop	12	3, <init>
$acc \leftarrow acc + top_of_stack$	15	3, <init>
pop	15	<init>

12

Notes

- It is very important that the stack is preserved across the evaluation of a subexpression
 - Stack before the evaluation of $7 + 5$ is $3, \langle \text{init} \rangle$
 - Stack after the evaluation of $7 + 5$ is $3, \langle \text{init} \rangle$
 - The first operand is on top of the stack

13

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

14

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 $\$sp + 4$

15

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

16

A Sample of MIPS Instructions

- lw reg_1 offset(reg_2) "load word"
 - Load 32-bit word from address $reg_2 + \text{offset}$ into reg_1
- add reg_1 reg_2 reg_3
 - $reg_1 \leftarrow reg_2 + reg_3$
- sw reg_1 offset(reg_2) "store word"
 - Store 32-bit word in reg_1 at address $reg_2 + \text{offset}$
- addiu reg_1 reg_2 imm "add immediate"
 - $reg_1 \leftarrow reg_2 + \text{imm}$
 - "u" means overflow is not checked
- li reg imm "load immediate"
 - $reg \leftarrow \text{imm}$

17

MIPS Assembly: Example

- The stack-machine code for $7 + 5$ in MIPS:

```
acc ← 7           li $a0 7
push acc          sw $a0 0($sp)
                  addiu $sp $sp -4
acc ← 5           li $a0 5
acc ← acc + top_of_stack lw $t1 4($sp)
                  add $a0 $a0 $t1
pop              addiu $sp $sp 4
```

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

18

A Small Language

- A language with only integers and integer operations ("**Mini Bar**")

$P \rightarrow F P \mid F$

$F \rightarrow \text{id}(\text{ARGS}) \text{ begin } E \text{ end}$

$\text{ARGS} \rightarrow \text{id}, \text{ARGS} \mid \text{id}$

$E \rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$
 $\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1, \dots, E_n)$

19

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:

```
fib(x)
begin
  if x = 1 then 0 else
  if x = 2 then 1 else fib(x - 1) + fib(x - 2)
end
```

20

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

21

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

22

Code Generation for Addition

```
 $cgen(e_1 + e_2) =$   
   $cgen(e_1)$            ;  $\$a0 \leftarrow$  value of  $e_1$   
  sw  $\$a0$  0( $\$sp$ )      ; push that value  
  addiu  $\$sp$   $\$sp$  -4   ; onto the stack  
   $cgen(e_2)$            ;  $\$a0 \leftarrow$  value of  $e_2$   
  lw  $\$t1$  4( $\$sp$ )      ; grab value of  $e_1$   
  add  $\$a0$   $\$t1$   $\$a0$     ; do the addition  
  addiu  $\$sp$   $\$sp$  4    ; pop the stack
```

Possible optimization:

Put the result of e_1 directly in register $\$t1$?

23

Code Generation for Addition: Wrong Attempt!

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

```
 $cgen(e_1 + e_2) =$   
   $cgen(e_1)$            ;  $\$a0 \leftarrow$  value of  $e_1$   
  move  $\$t1$   $\$a0$        ; save that value in  $\$t1$   
   $cgen(e_2)$            ;  $\$a0 \leftarrow$  value of  $e_2$   
                               ; may clobber  $\$t1$   
  add  $\$a0$   $\$t1$   $\$a0$     ; perform the addition
```

Try to generate code for : $3 + (7 + 5)$

24

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

25

Code Generation for Subtraction and Constants

New instruction: `sub reg1 reg2 reg3`

Implements $reg_1 \leftarrow reg_2 - reg_3$

```
cgen(e1 - e2) =  
  cgen(e1)           ; $a0 ← value of e1  
  sw $a0 0($sp)       ; push that value  
  addiu $sp $sp -4    ; onto the stack  
  cgen(e2)           ; $a0 ← value of e2  
  lw $t1 4($sp)       ; grab value of e2  
  sub $a0 $t1 $a0      ; do the subtraction  
  addiu $sp $sp 4     ; pop the stack
```

26

Code Generation for Conditional

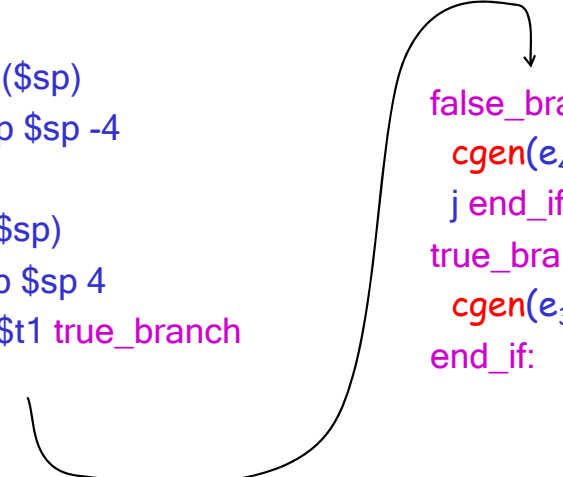
- We need flow control instructions
- New MIPS instruction: `beq reg1 reg2 label`
 - Branch to `label` if $reg_1 = reg_2$
- New MIPS instruction: `j label`
 - Unconditional jump to `label`

27

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)  
  j end_if  
true_branch:  
  cgen(e3)  
end_if:
```



28

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 $f(x_1, \dots, x_n)$ push the arguments x_n, \dots, x_1 onto the stack
 - These are the only variables in this language

29

Meet The Activation Record (Cont.)

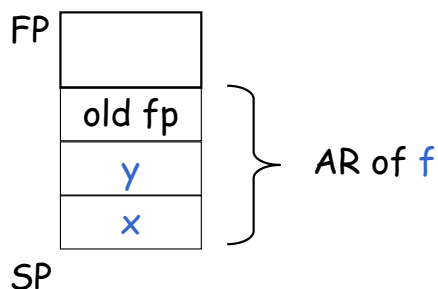
- The stack discipline guarantees that on function exit, $\$sp$ is the same as it was before the args got pushed (i.e., before function call)
- We need the return address
- It's also handy to have a pointer to the current activation
 - This pointer lives in register $\$fp$ (frame pointer)
 - Reason for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

30

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:



31

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 special register $\$ra$
 - On other architectures the return address is stored on the stack by the "call" instruction

32

Code Generation for Function Call (Cont.)

$cgen(f(e_1, \dots, e_n)) =$

```
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
```

- 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 $4*n+4$ bytes long

33

Code Generation for Function Definition

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

$cgen(f(x_1, \dots, x_n) \text{ begin } e \text{ end}) =$

`f_entry:`

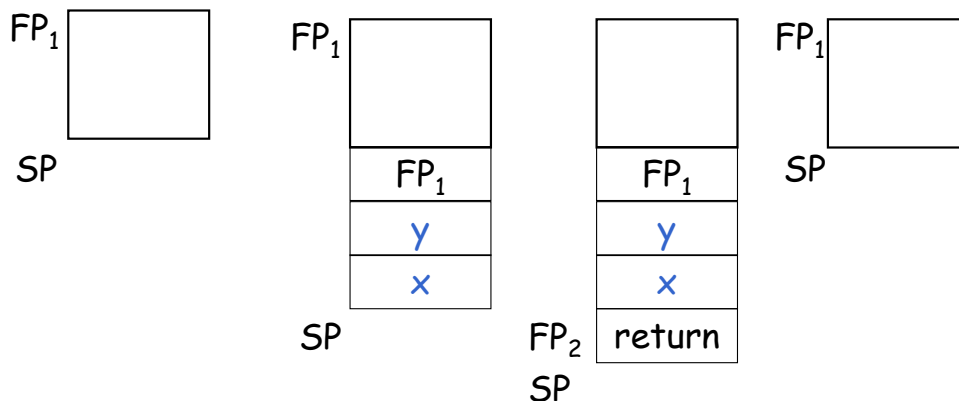
```
move $fp $sp
sw $ra 0($sp)
addiu $sp $sp -4
cgen(e)
lw $ra 4($sp)
addiu $sp $sp frame_size
lw $fp 0($sp)
jr $ra
```

- Note: The frame pointer points to the top, not bottom of the frame
- Callee saves old return addr, evaluates its body, pops the return addr, pops the args, and then restores `$fp`
- $frame_size = 4*n + 8$

34

Calling Sequence: Example for $f(x,y)$

Before call On entry After body After call



35

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`

36

Code Generation for Variables/Parameters

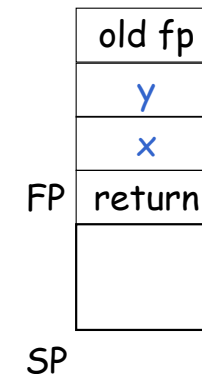
- Solution: use the frame pointer
 - Always points to the return address on the stack
 - Since it does not move, it can be used to find the variables
- Let x_i be the i^{th} ($i = 1, \dots, n$) formal parameter of the function for which code is being generated

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

37

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):



- x is at $fp + 4$
- y is at $fp + 8$

38

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

39

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

40