Code Generation

Outline

- Stack machines
- The MIPS assembly language
- A simple source language
- Stack-machine implementation of the simple language

Stack Machines

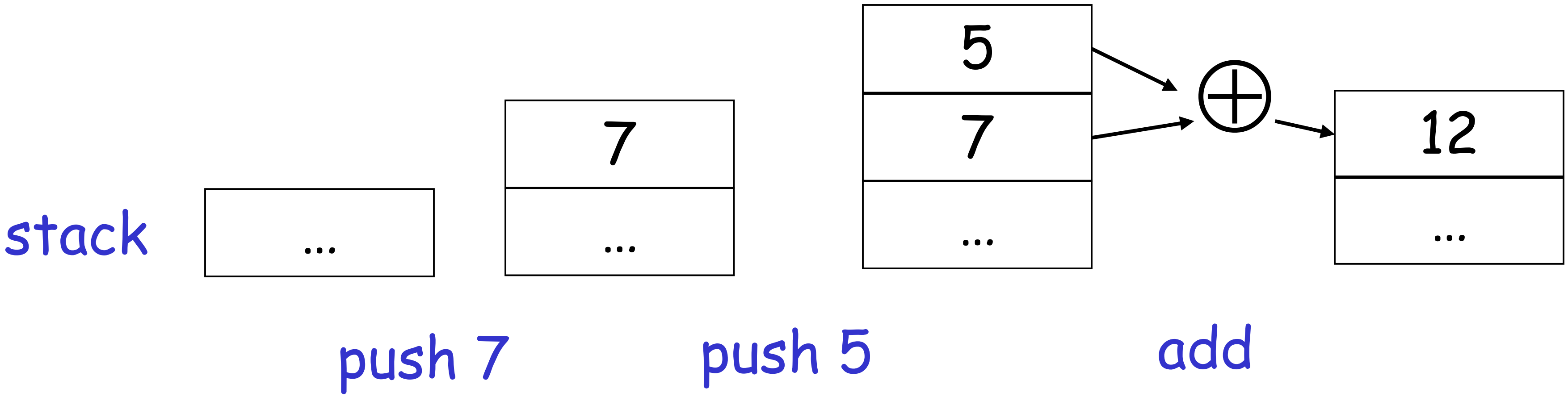
- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results

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
```

Stack Machine. Example



- 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

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 r_1, r_2"
  - ⇒ Smaller encoding of instructions
  - ⇒ More compact programs
- This is one reason why the Java Virtual Machine 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 most recently computed value in a register (called accumulator) since register accesses are faster.
- The "add" instruction is now

  $$acc \leftarrow acc + top\_of\_stack$$

  - Only one memory operation!

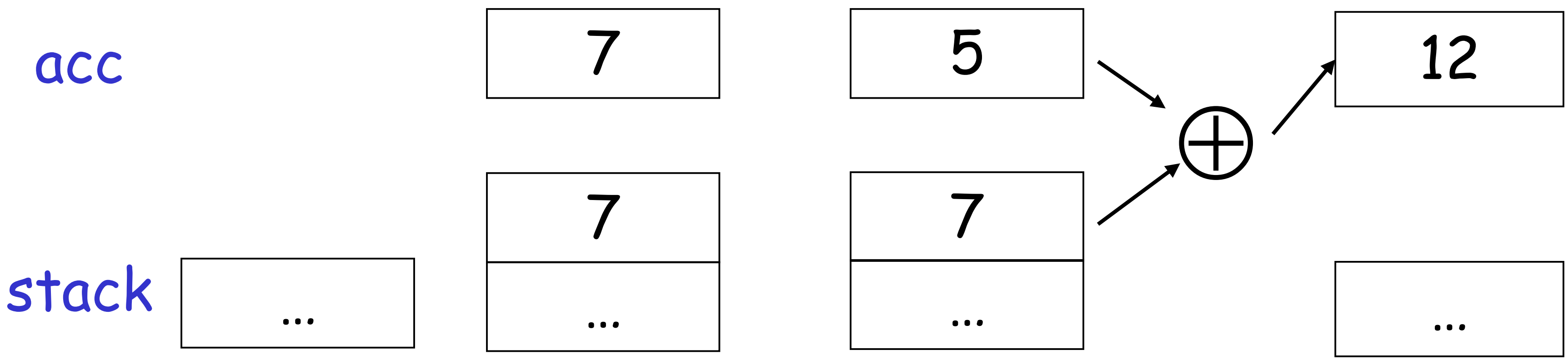
Stack Machine with Accumulator

Invariants

- The result of computing an expression is always in the accumulator
- For an operation $op(e_1,\ldots,e_n)$ push the accumulator on the stack after computing each of $e_1,\ldots,e_{n-1}$
  - The result of e_n is in the accumulator before op
  - 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 ← 7
push acc

acc ← 5

acc ← acc + top_of_stack
pop

A Bigger Example: 3 + (7 + 5)

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

From Stack Machines to MIPS

- The compiler generates code for a stack machine with accumulator
- We want to run the resulting code on an x86 or MIPS processor (or simulator)
- We implement stack machine instructions using MIPS instructions and registers

Why use MIPS assembly

- it's somewhat more readable than x86 assembly
- using a MIPS simulator is simpler

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 both MIPS and x86
- 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

- Typical 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 reg_1, offset(reg_2)
  - Load 32-bit word from address reg_2 + 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 32-bit word in reg_1 at address reg_2 + offset
- addiu reg_1, reg_2, imm
  - $reg_1 \leftarrow reg_2 + imm$
  - "u" means overflow is not checked
- li reg, imm
  - $reg \leftarrow imm$

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…

A Small Language

- A language with integers and integer operations

$$
\begin{aligned}
P &\rightarrow D; P \mid D \\
D &\rightarrow \text{def id(ARGS) = E;} \\
ARGS &\rightarrow \text{id, ARGS} \mid \text{id} \\
E &\rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4 \\
&\quad \mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)
\end{aligned}
$$

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

- 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

Some Useful Macros

- We define the following abbreviation
- push $t   sw $t, 0($sp)
  addiu $sp, $sp, -4
- pop   addiu $sp, $sp, 4
- $t $\leftarrow$ top   lw $t, 4($sp)

Code Generation for Constants

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

  cgen(i) = li $a0, i

- This also preserves the stack, as required

Code Generation for Add

$cgen(e_1 + e_2) =$

```
        cgen(e1)
        push $a0
        cgen(e2)
        $t1 ← top
        add $a0, $t1, $a0
        pop
```

- 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?

  $cgen(e_1 + e_2) =$

  $\quad cgen(e_1)$

  $\quad$ move \$t1, \$a0

  $\quad cgen(e_2)$

  $\quad$ 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 (modified) post-order traversal of the AST, at least for expressions

Code Generation for Sub and Constants

- New instruction: sub reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

```
cgen(e1 - e2) =
        cgen(e1)
        push $a0
        cgen(e2)
        $t1 ← top
        sub $a0, $t1, $a0
        pop
```

Code Generation for Conditional

- We need flow control instructions

- New instruction: beq reg_1, reg_2, label
  - Branch to label if $reg_1 = reg_2$

- New instruction: j label
  - Unconditional jump to label

Code Generation for If (Cont.)

cgen(if $e_1 = e_2$ then e_3 else e_4) =
  false_branch = new_label ()
  true_branch = new_label ()
  end_if = new_label ()
  cgen(e_1)
  push $a0
  cgen(e_2)
  $t1 ← top
  pop
  beq $a0, $t1, true_branch

false_branch:
  cgen(e_4)
  b end_if
true_branch:
  cgen(e_3)
end_if: