Code Generation

Lecture 31
(courtesy R. Bodik)

Lecture Outline

- Stack machines
- The MIPS assembly language
- The x86 assembly language
- A simple source language
- Stack-machine implementation of the simple language

Stack Machines

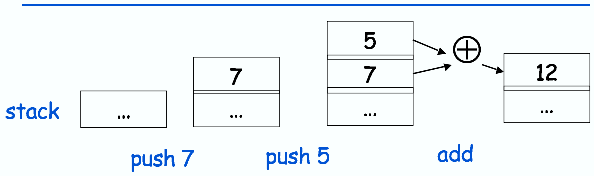
- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results

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 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!

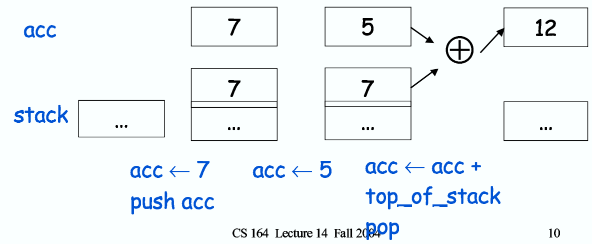
Stack Machine with Accumulator

Invariants

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



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>

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 an x86 or MIPS processor (or simulator)
- We implement stack machine instructions using MIPS instructions and registers

MIPS assembly vs. x86 assembly

- In PA4 and PA5, you will generate x86 code
  - because we have no MIPS machines around
  - and using a MIPS simulator is less exciting
- In this lecture, we will use MIPS assembly
  - it's somewhat more readable than x86 assembly
  - e.g. in x86, both store and load are called movl
- translation from MIPS to x86 trivial
  - see the translation table in a few slides

Simulating a Stack Machine…

- The accumulator is kept in MIPS register $a0
  - in x86, it's in %eax
- 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
  - in x86, its' %esp

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 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 ← imm

x86 Assembly

x86 architecture

- Complex Instruction Set Computer (CISC) architecture
- Arithmetic operations can use both registers and memory for operands and results
- So, you don't have to use separate load and store instructions to operate on values in memory
- CISC gives us more freedom in selecting instructions (hence, more powerful optimizations)
- but we'll use a simple RISC subset of x86
  - so translation from MIPS to x86 will be easy

x86 assembly

- x86 has two-operand instructions:
  - ex.: ADD dest, src   dest := dest + src
  - in MIPS: dest := src1 + src2
- An annoying fact to remember ☹
  - different x86 assembly versions exists
  - one important difference: order of operands
  - the manuals assume
    - ADD dest, src
  - the gcc assembler we'll use uses opposite order
    - ADD src, dest

Sample x86 instructions (gcc order of operands)

- movl offset(reg_2), reg_1
  - Load 32-bit word from address reg_2 + offset into reg_1
- add reg_2, reg_1
  - $reg_1 \leftarrow reg_1 + reg_2$
- movl reg_1 offset(reg_2)
  - Store 32-bit word in reg_1 at address reg_2 + offset
- add imm, reg_1
  - $reg_1 \leftarrow reg_1$ + imm
  - use this for MIPS' addiu
- movl imm, reg
  - reg ← imm

MIPS to x86 translation

MIPS	x86
lw reg_1, offset(reg_2)	movl offset(reg_2), reg_1
add reg_1, reg_1, reg_2	add reg_2, reg_1
sw reg_1, offset(reg_2)	movl reg_1, offset(reg_2)
addiu reg_1, reg_1, imm	add imm, reg_1
li reg, imm	movl imm, reg

x86 vs. MIPS registers

MIPS	x86
$a0	%eax
$sp	%esp
$fp	%ebp
$t	%ebx

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…

Some Useful Macros

- We define the following abbreviation
- push $t   addiu $sp, $sp, -4
                   sw $a0, 0($sp)
- pop   addiu $sp, $sp, 4
- $t ← top   lw $t, 0($sp)

A Small Language

- A language with integers and integer operations

$P \rightarrow D; P \mid D$

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

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

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

$\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)$

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

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(e1 + e2) =
      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(e1 + e2) =
      cgen(e1)
      move $t1, $a0
      cgen(e2)
      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 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 $reg_1 = reg_2$
  - x86: cmpl reg_1, reg_2
    je label
- New instruction: b label
  - Unconditional jump to label
  - x86: jmp label

Code Generation for If (Cont.)

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

```
  cgen(e1)
  push $a0
  cgen(e2)
  $t1 ← top
  pop
  beq $a0, $t1, true_branch
false_branch:
  cgen(e4)
  b end_if
true_branch:
  cgen(e3)
end_if:
```