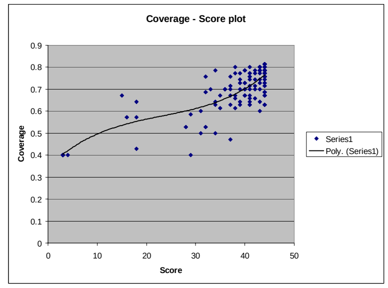
Code Generation

Lecture 12

The remote testing experiment. It works!



Remote testing

- In PA1, if you achieved best coverage, you also got best score!

44	0.814286
44	0.814286
44	0.814286
44	0.814286
43	0.8
43	0.8
43	0.8
43	0.8
44	0.8
41	0.8
38	0.8
44	0.8

From the cs164 newsgroup ☺

› I need a small website made. I'm willing to pay for the work too.
› So... if anyone is interested e-mail me at [deleted]@berkeley.edu.

CS164's guide to create a website:
1) Write a lexer
2) Write a lexer spec for some unknown, obscure language
3) Write a parser
4) Write the grammar for that obscure language
5) Write a code generator that generates HTML
6) ...
7) Profit! Now only you can maintain that website!

The moral

- Essentially, the recommended strategy is to
  - **goal:** no one can maintain your programs
  - **means:** develop an obscure language for your programs
- But if this is your goal, why a new language?
  - tons of unmaintainable Java programs written
  - some even submitted as cs164 projects ☺
  - I am sure you can succeed with just Java, too.
- A better road to profit
  - develop a language: can be obscure, even horrible, but make sure it's horibly useful, too (ex.: perl, C++, Visual Basic, latex)
  - then publish books on this language ☺

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

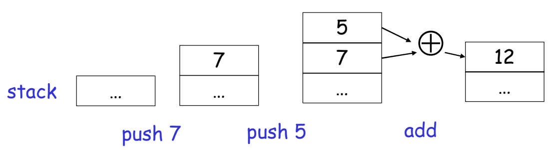
- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results

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!

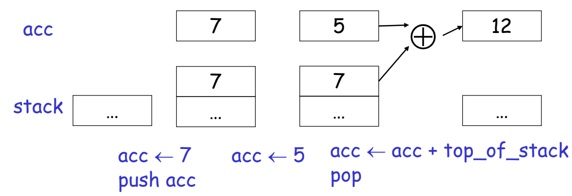
Stack Machine with Accumulator

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
  - $reg \leftarrow imm$

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
  - $reg \leftarrow imm$

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 — sw $a0 0($sp)
  addiu $sp $sp -4
- pop — addiu $sp $sp 4
- $t ← top — lw $t 4($sp)

A Small Language

- A language with integers and integer operations

P → D; P | D

D → def id(ARGS) = E;

ARGS → id, ARGS | id

E → int | id | if E_1 = E_2 then E_3 else E_4

| E_1 + E_2 | E_1 - E_2 | id(E_1,…,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

$cgen(e_1 + e_2)$ =
- $cgen(e_1)$
- push $a0
- $cgen(e_2)$
- $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?

$cgen(e_1 + e_2)$ =
- $cgen(e_1)$
- 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 reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

$cgen(e_1 - e_2)$ =
- $cgen(e_1)$
- push $a0
- $cgen(e_2)$
- $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(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:

The Activation Record

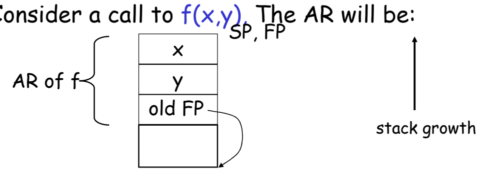
- Code for function calls and function definitions depends on the layout of the activation record
- A very simple AR suffices for this language:
  - The result is always in the accumulator
    - No need to store the result in the AR
  - The activation record holds actual parameters
    - For $f(x_1,\dots,x_n)$ push $x_n,\dots,x_1$ 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 to save $sp
- We need the return address
- It's handy to have a pointer to start of the current activation
  - 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
- Consider a call to f(x,y), The AR will be:



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 $ra
  - x86: the return address is stored on the stack by the call label instruction

Code Generation for Function Call (Cont.)

```
cgen(f(e1,…,en)) =
    push $fp
    cgen(en)
    push $a0
    …
    cgen(e1)
    push $a0
    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 4*n+4 bytes long

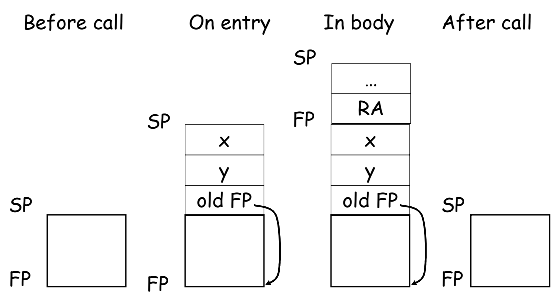
Code Generation for Function Definition

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

```
cgen(def f(x1,…,xn) = e) =
    move $fp $sp
    push $ra
    cgen(e)
    $ra ← top
    addiu $sp $sp z
    lw $fp 0($sp)
    jr $ra
```

- 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).



Code Generation for Variables

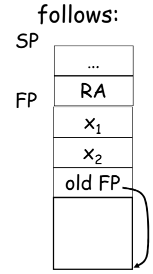
- Variable references are the last construct
- The "variables" of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller
- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp

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

Code Generation for Variables (Cont.)

- Example: For a function def f(x_1,x_2) = e the activation and frame pointer are set up as follows:



x_1 is at fp + 4

x_2 is at fp + 8

- Thus:

cgen(x_i) = lw $a0 z($fp)

(z = 4*i)

Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- We recommend you use a stack machine for your Decaf compiler (it's simple)

Summary

- See the PA4 starter kit for a large code generation example
- 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

Allocating Temporaries in the AR

Review

- The stack machine has activation records and intermediate results interleaved on the stack

AR
Intermediates
AR
Intermediates

Review (Cont.)

- Advantage: Very simple code generation
- Disadvantage: Very slow code
  - Storing/loading temporaries requires a store/load and $sp adjustment

A Better Way

- Idea: Keep temporaries in the AR
- The code generator must assign a location in the AR for each temporary

Example

def fib(x) = if x = 1 then 0 else
if x = 2 then 1 else
fib(x - 1) + fib(x – 2)

- What intermediate values are placed on the stack?
- How many slots are needed in the AR to hold these values?

How Many Temporaries?

- Let NT(e) = # of temps needed to evaluate e
- NT(e_1 + e_2)
  - Needs at least as many temporaries as NT(e_1)
  - Needs at least as many temporaries as NT(e_2) + 1
- Space used for temporaries in e_1 can be reused for temporaries in e_2

The Equations

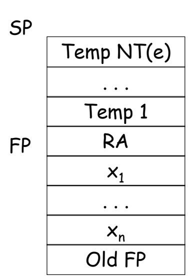
NT(e_1 + e_2) = max(NT(e_1), 1 + NT(e_2))
NT(e_1 - e_2) = max(NT(e_1), 1 + NT(e_2))
NT(if e_1 = e_2 then e_3 else e_4) = max(NT(e_1),1 + NT(e_2), NT(e_3), NT(e_4))
NT(id(e_1,…,e_n) = max(NT(e_1),…,NT(e_n))
NT(int) = 0
NT(id) = 0

Is this bottom-up or top-down?
What is NT(…code for fib…)?

The Revised AR

- For a function definition f(x_1,…,x_n) = e the AR has 2 + n + NT(e) elements
  - Return address
  - Frame pointer
  - n arguments
  - NT(e) locations for intermediate results

Picture



Revised Code Generation

- Code generation must know how many temporaries are in use at each point
- Add a new argument to code generation: the position of the next available temporary

Code Generation for + (original)

```
cgen(e1 + e2) =
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
        cgen(e2)
        lw $t1 4($sp)
        add $a0 $t1 $a0
        addiu $sp $sp 4
```


Code Generation for + (revised)

```
cgen(e1 + e2, nt) =
        cgen(e1, nt)
        sw $a0 -nt($fp)
        cgen(e2, nt + 4)
        lw $t1 -nt($fp)
        add $a0 $t1 $a0
```


Notes

- The temporary area is used like a small, fixed-size stack
- Exercise: Write out cgen for other constructs

Code Generation for Object-Oriented Languages

Object Layout

- OO implementation = Stuff from last lecture + More stuff
- OO Slogan: If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected
- This means that code in class A works unmodified for an object of class B

Two Issues

- How are objects represented in memory?
- How is dynamic dispatch implemented?

Object Layout Example

```
class A {
    int a = 0;
    int d = 1;
    int f() { return a = a + d }
}

class B extends A {
    int b = 2;
    int f() { return a } //override
    int g() { return a = a - b }
}

class C extends A {
    int c = 3;
    int h() { return a = a * c }
}
```


Object Layout (Cont.)

- Attributes a and d are inherited by classes B and C
- All methods in all classes refer to a
- For A methods to work correctly in A, B, and C objects, attribute a must be in the same "place" in each object

Object Layout (Cont.)

An object is like a struct in C. The reference

foo.field

is an index into a foo struct at an offset corresponding to field

- Objects are laid out in contiguous memory
- Each attribute stored at a fixed offset in object
- When a method is invoked, the object is this and the fields are the object's attributes

Object Layout

- The first 3 words of an object contain header information:

	Offset
Class Tag	*0*
Object Size	*4*
Dispatch Ptr	*8*
Attribute 1	*12*
Attribute 2	*16*
. . .	

Object Layout (Cont.)

- Class tag is an integer
  - Identifies class of the object
- Object size is an integer
  - Size of the object in words
- Dispatch ptr is a pointer to a table of methods
  - More later
- Attributes in subsequent slots
- Lay out in contiguous memory

Subclasses

Observation: Given a layout for class A, a layout for subclass B can be defined by extending the layout of A with additional slots for the additional attributes of B

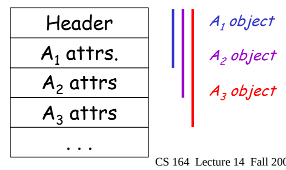
Leaves the layout of A unchanged
(B is an extension)

Layout Picture

Offset / Class	0	4	8	12	16	20
A	Atag	5	*	a	d	
B	Btag	6	*	a	d	b
C	Ctag	6	*	a	d	c

Subclasses (Cont.)

- The offset for an attribute is the same in a class and all of its subclasses
  - Any method for an A_1 can be used on a subclass A_2
- Consider layout for $A_n \leq \ldots \leq A_3 \leq A_2 \leq A_1$



What about multiple inheritance?

Dynamic Dispatch

- Consider again our example

```
class A {
    int a = 0;
    int d = 1;
    int f() { return a = a + d }
}

class B extends A {
    int b = 2;
    int f() { return a }
    int g() { return a = a - b }
}

class C extends A {
    int c = 3;
    int h() { return a = a * c }
}
```


Dynamic Dispatch Example

- e.g()
  - g refers to method in B if e is a B
- e.f()
  - f refers to method in A if f is an A or C (inherited in the case of C)
  - f refers to method in B for a B object
- The implementation of methods and dynamic dispatch strongly resembles the implementation of attributes

Dispatch Tables

- Every class has a fixed set of methods (including inherited methods)
- A *dispatch table* indexes these methods
  - An array of method entry points
  - A method f lives at a fixed offset in the dispatch table for a class **and all of its subclasses**

Dispatch Table Example

Offset / Class	0	4
A	fA	
B	fB	g
C	fA	h

- The dispatch table for class A has only 1 method
- The tables for B and C extend the table for A to the right
- Because methods can be overridden, the method for f is not the same in every class, but is always at the same offset

Using Dispatch Tables

- The dispatch pointer in an object of class X points to the dispatch table for class X
- Every method f of class X is assigned an offset O_f in the dispatch table at compile time

Using Dispatch Tables (Cont.)

- Every method must know what object is "this"
  - "this" is passed as the first argument to all methods
- To implement a dynamic dispatch e.f() we
  - Evaluate e, obtaining an object x
  - Find D by reading the dispatch-table field of x
  - Call $D[O_f](x)$
    - D is the dispatch table for x
    - In the call, this is bound to x