CS 412/413

Introduction to
Compilers and Translators

Cornell University
Andrew Myers

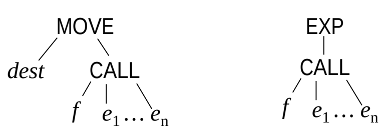
Lecture 17: Finishing basic code generation
3 March 00

Outline

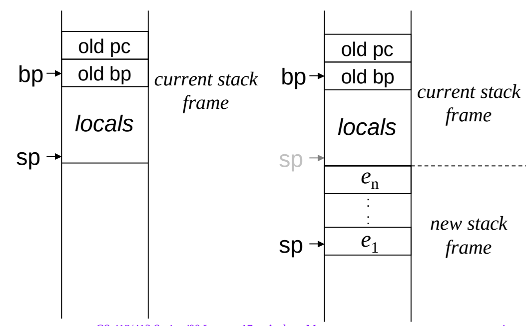
- Implementing function calls
- Implementing functions
- Optimizing away the frame pointer
- Dynamically-allocated structures: strings and arrays
- Register allocation the easy way

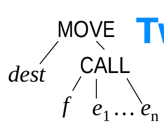
Function calls

- How to generate code for function calls?
- Two kinds of IR statements in canonical form

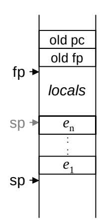


Stack layout





Two translations



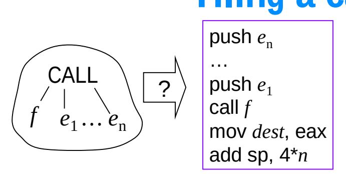
non-RISC

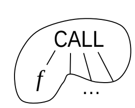
```
push en
…
push e1
call f
mov dest, eax
add sp, 4*n
```

RISC

```
sub sp, 4*n
mov [sp + 4], e1
...
mov [sp + 4*n], en
call f
mov dest, eax
add sp, 4*n
```

Tiling a call





- Problem: doesn't fit into tiling paradigm; unbounded # tiles required
- Solution: don't fold e_i into tile
- Downside: generates a lot of extra temporaries

```
push tn
…
push t1
call f
mov dest, eax
add sp, 4*n
```

Compiling function bodies

- Function body:

  S ⟦ f (...) : T = e ⟧ = MOVE(RV, E ⟦ e ⟧)

  S ⟦ f (...) = e ⟧ = EXP(E ⟦ e ⟧)
- Variables:

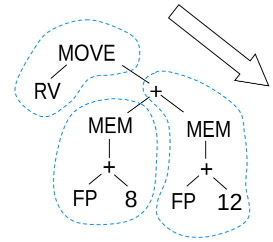
  E ⟦ v ⟧ = TEMP(t_v) for locals

  E ⟦ v ⟧ = MEM(TEMP(FP) + k) for args
- Try it out:

  f(x: int, y:int) = x + y

Abstract assembly for f

f(x: int, y:int) = x + y

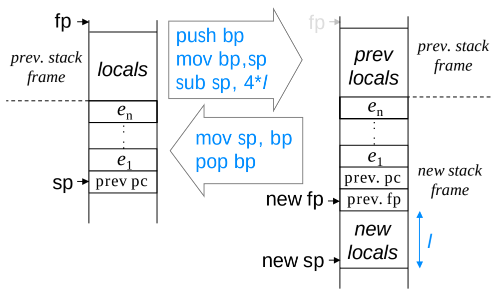



```
mov t1, [ebp+8]
mov t2, t1
add t2, [ebp+12]
mov eax, t2
```

What's missing here?

Stack frame setup

- Need code to set up stack frame on entry



Function code

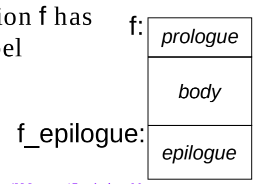
```
f: push bp
   mov bp,sp          } function prologue
   sub sp, 4*l
   mov t1, [ebp+8]
   mov t2, t1
   add t2, [ebp+12]
   mov eax, t2
f_epilogue:
   mov sp, bp
   pop bp             } function epilogue
   ret
```


Compiling return

- Iota return statement returns immediately from function. Translation:

S ⟦ return e ⟧ = SEQ(MOVE(RV, E ⟦ e ⟧),
JUMP(*epilogue*))

- Every function f has epilogue label



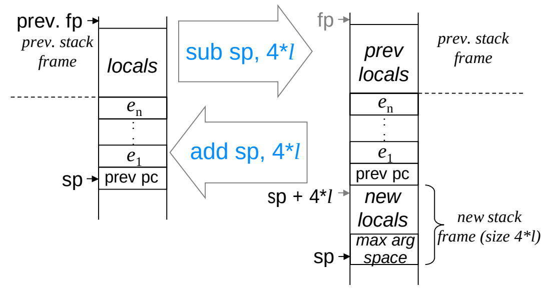
The Glory of Pentium CISC

```
f:  push ebp              enter 4*l, 0
    mov ebp,esp
    sub esp, 4*l
    mov t1, [ebp+8]
    mov t2, t1
    add t2, [ebp+12]
    mov eax, t2
f_epilogue:               f_epilogue:
    mov esp, ebp              leave
    pop ebp                   ret
    ret
```

Optimizing away ebp

- Idea: maintain constant offset *k* between frame pointer and stack pointer
- Use RISC-style argument passing rather than pushing arguments on stack
- All references to MEM(FP+n) translated to operand `[esp+(n+k)]` instead of to `[ebp+n]`
- Advantage: get whole extra register to use when allocating registers (7!)

Stack frame setup



Caveats

- Get even faster (and RISC-core) prologue and epilogue than with **`enter/leave`** but:
- Must save **ebp** register if we want to use it (like **esp**, callee-save)
- Doesn't work if stack frame is truly variable-sized : *e.g.*, alloca() call in C allocates variable-sized array *on the stack* -- not a problem in Iota where arrays heap-allocated

Dynamic structures

- Modern programming languages allow dynamically allocated data structures: strings, arrays, objects

C: char *x = (char *)malloc(strlen(s) + 1);

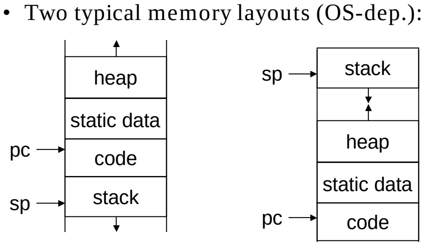
C++: Foo *f = new Foo(…);

Java: Foo f = new Foo(…);
String s = s1 + s2;

Iota: x: array[int] = new int[5] (0);
String s = s1 + s2;

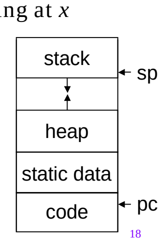
Program Heap

- Program has 4 memory areas: code segment, stack segment, static data, heap
- Two typical memory layouts (OS-dep.):



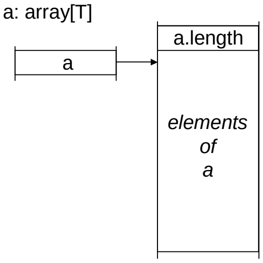
Object allocation

- Dynamic objects allocated in the heap
  - array creation, string concatenation
  - malloc(*n*) returns new chunk of *n* bytes, free(*x*) releases memory starting at *x*
- Globals statically allocated in data segment
  - global variables
  - string constants
  - assembler supports data segment declarations



Iota dynamic structures

a: array[T]



new *T* [*n*] (*e*)

⇩

a = malloc(4**n* + 4);
MEM(a) = *n*;
a = a + 4;
a_1 = a; a_2 = a + 4**n*;
while (a_1 != a_2)
  (MEM(a_1) = E ⟦e⟧;
   a_1 = a_1 + 4);
a

- PA4: We give you newarray, newstring calls with garbage collection support (no free call needed)

Trivial register allocation

- Can convert abstract assembly to real assembly easily (but generate bad code)
- Allocate every temporary to location in the current stack frame rather than to a register
- Every temporary stored in different place -- no possibility of conflict
- Three registers needed to shuttle data in and out of stack frame (max. # registers used by one instruction) : *e.g*, eax, ebx, ecx

Rewriting abstract code

- Given instruction, replace every temporary in instruction with one of three registers
- Add mov instructions before instruction to load registers properly
- Add mov instructions after instruction to put data back onto stack (if necessary)

push t1 ⇒ mov eax, [fp - t1off]; push eax
mov [fp+4], t3 ⇒ ?
add t1, [fp - 4] ⇒ ?

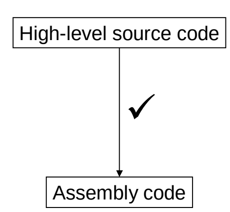
Result

- Simple way to get working code
- Code is longer than necessary, slower
- Also can allocate temporaries to registers until registers run out (3 temporaries on Pentium, 20+ on MIPS, Alpha)
- Code generation technique actually used by some compilers when all optimization turned off (-O0)
- Will use for Programming Assignment 4

Summary

- Complete code generation technique
- Use tiling to perform instruction selection
- Arguments mapped to stack locations, locals to temporaries
- Function code generated by gluing prologue, epilogue onto body
- Dynamic structure allocation handled by relying on heap allocation routines (malloc)
- Static structures allocated by data segment assembler declarations
- Allocate temporaries to stack locations to eliminate use of unbounded # of registers
- Shuttle temporaries in and out using eax-ecx regs

Where we are



Further topics

- Generating better code
  - register allocation
  - optimization (high- and low-level)
  - dataflow analysis
- Supporting language features
  - objects, modules, polymorphism, first-class functions, exceptions
  - advanced GC techniques
  - dynamic linking, loading, & PIC
  - dynamic types and reflection
- Compiler-like programs
  - source-to-source translation
  - interpreters