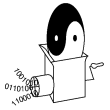


# CS 4120
# Introduction to Compilers

Andrew Myers
Cornell University

Lecture 18: Finishing code generation

7 Oct 09

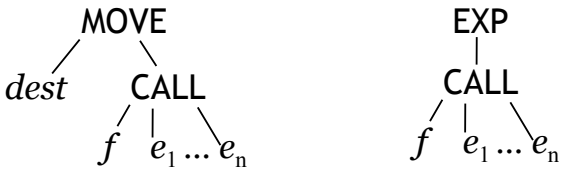
# Outline

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



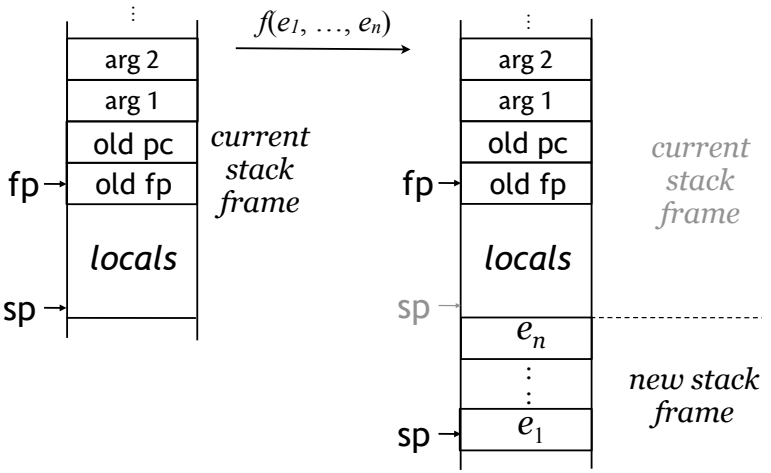
# Function calls

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

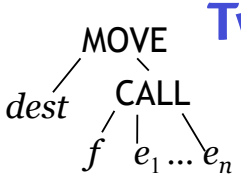


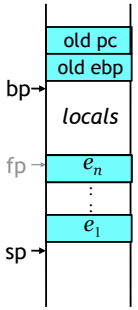



# Stack layout



# Two translations




*non-RISC*

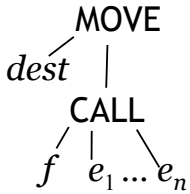
```
push e_n
...
push e_1
call f
mov dest, eax
add esp, 4*n
```

*RISC*

```
sub esp, 4*n
mov [esp + 4], e_1
...
mov [esp + 4*n], e_n
call f
mov dest, eax
add esp, 4*n
```



# Tiling a call




```
push e_n
...
push e_1
call f
mov dest, eax
add esp, 4*n
```

Try to match $e_i$ against r/m32, tile $e_i$ separately if failure.

Push is non-RISC, but still fast on IA-32.



# Compiling function bodies

- How we compiled function body:

$\mathcal{S}[\![ f(\ldots) : T = e ]\!]$ = SEQ(MOVE(RV, $\mathcal{E}[\![ e ]\!]$), LABEL(*epilogue*))

$\mathcal{S}[\![ f(\ldots) = e ]\!]$ = SEQ(EXP($\mathcal{E}[\![ e ]\!]$), LABEL(*epilogue*))

- Variables:

→ $\mathcal{E}[\![ v ]\!]$ = TEMP($t_v$) for locals

→ $\mathcal{E}[\![ v ]\!]$ = MEM(TEMP(FP) + (4**i*+4)) for arg. *i*

→ $\mathcal{E}[\![ v ]\!]$ = MEM(NAME($v$)) for globals

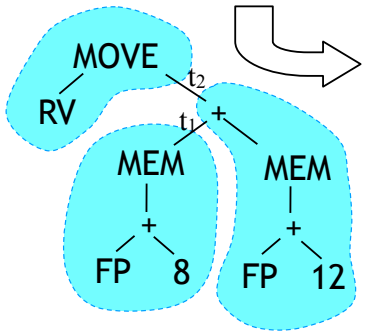
- Try it out:

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



# Abstract assembly for f

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

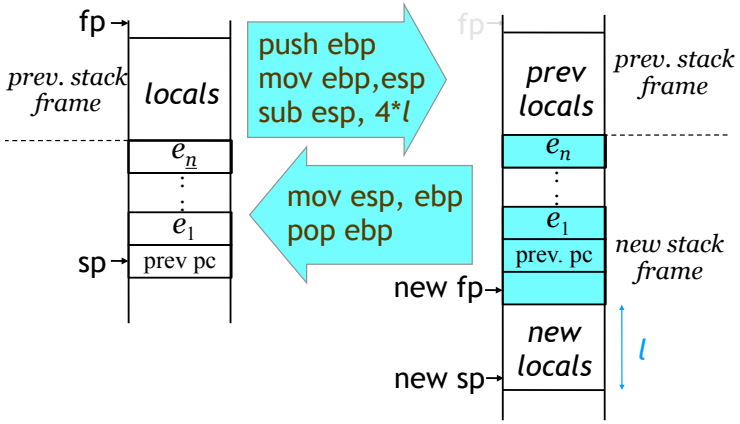



```
mov t1, [ebp+8]
mov t2, t1
add t2, [ebp+12]
mov eax, t2
```

What's missing here?

# Stack frame setup

- Need code to set up stack frame on entry






# Function code

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

function *prologue*

function *epilogue*



# The Glory of CISC

```
f: push ebp
   mov ebp,esp
   sub esp, 4*l
```

`enter 4*l, 0`

```
   mov t1, [ebp+8]
   mov t2, t1
   add t2, [ebp+12]
   mov eax, t2
f_epilogue:
   mov esp, ebp
   pop ebp
   ret
```

```
f_epilogue:
   leave
   ret
```



# Calling conventions

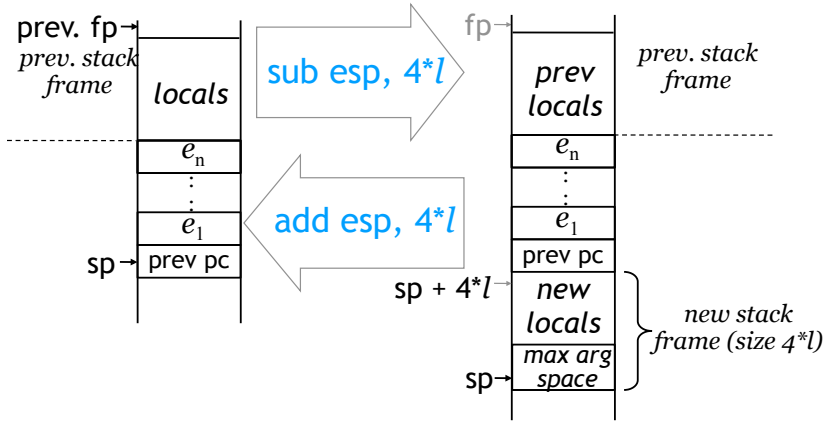
- Have described the C calling convention:
  - arguments pushed on stack, in reverse order.
  - caller is responsible for cleaning up arguments.
- Others:
  - stdcall: callee cleans up.
  - MIPS/Alpha: first 4/6 arguments passed in registers.
  - fastcall: first 2 arguments passed in ecx, edx, rest on stack
  - See website for more details.
- Choice of caller- vs. callee-save:
  - **caller-save**: caller must save registers if needed after call; usable freely by callee (`e[acd]x`)
  - **callee-save**: callee not allowed to change; to use, must be saved (`e[sd]i`, `e[sb]p`, `ebx`)

# Optimizing away ebp

- Really no need for frame pointer register!
- 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: 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:
- To avoid breaking callers, must save **`ebp`** register if we want to use it (ebp is callee-save)
- Doesn't work if stack frame is truly variable-sized
  - alloca($n$) call in C allocates $n$ bytes *on the stack* – compiler cannot predict $n$
  - not a problem in Iota$_9$: arrays heap-allocated, stack frame has constant size



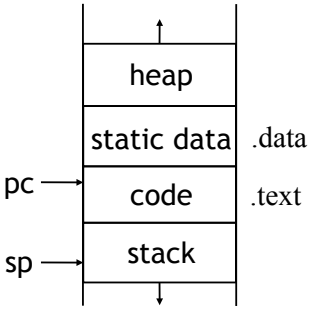
# Dynamic structures

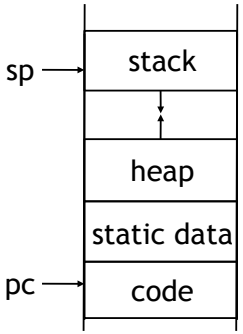
- Modern programming languages allow dynamically allocated data structures: strings, arrays, objects:

**C:** char *x = (char *)malloc(strlen(s) + 1);
**C++:** Foo *f = new Foo(…);
**Java:** Foo f = new Foo(…);
String s = s1 + s2;
**Iota:** x: int[5];
y: int[] = "hello"

# Program Heap

- Program has 4 memory areas: code segment, stack segment, static data, heap
- Two typical *virtual* memory layouts (depends on OS):

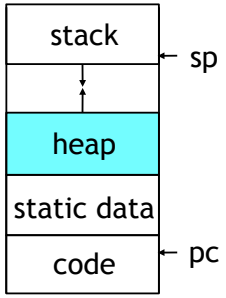




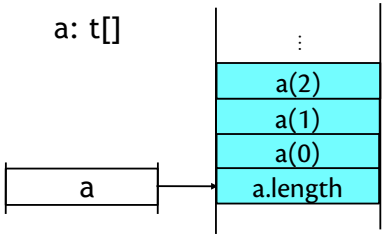

# Object allocation

- 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 has data segment declaration (.data)






# Iota dynamic structures




a: *t* [*n*]

a = CALL(malloc, 4**n* + 4);
MEM(a) = *n*;
a + 4;

- PA4: We give you malloc call 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
  - t1 = [ebp-4], t2=[ebp-8], t3 = [ebp-12], ...
- 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, ecx, edx

# Rewriting abstract code

- Given instruction, replace every temporary in instruction with one of three registers **`e[acd]x`**
- 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, [ebp-4]; push eax

mov [fp+4], t3 ⇒ ?



# 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

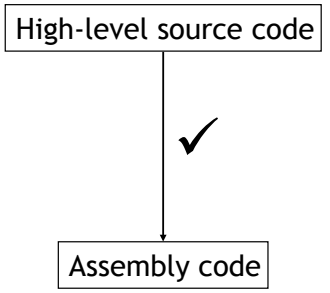


# Summary

- 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 e[acd]x regs



# Where we are



# Tiling, formally

- Each tile is really an inference rule!
- Write $e \Rightarrow c$ (reg $t$)

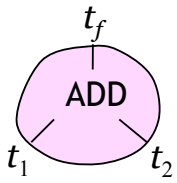
  "$c$ is valid assembly for IR $e$, puts result into $t$"

  – Like $\mathcal{E}[\![e]\!]$=$c$, but not a function
- Write $s \Rightarrow c$

  "$c$ is valid code for IR statement $s$"
- Translation is not a function of $e$/$s$ : many possible translations (unlike $\mathcal{E}[\![e]\!]$, $\mathcal{S}[\![e]\!]$,$\mathcal{T}[\![e]\!]$, etc.)
  - translation *relation* generalizes translation function
  - can apply idea to earlier translations too – but less payoff



# Expression tile as rule



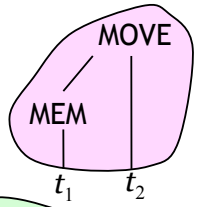

**mov** $t_f$, $t_1$
**add** $t_f$, $t_2$

$$\frac{e_1 \Rightarrow c_1\,(\text{reg } t_1) \quad e_2 \Rightarrow c_2\,(\text{reg } t_2)}{\text{ADD}(e_1, e_2) \Rightarrow c_1;\ c_2;\ \textbf{mov}\ t_f,\ t_1;\ \textbf{add}\ t_f,\ t_2 \quad (\text{reg } t_f)}$$

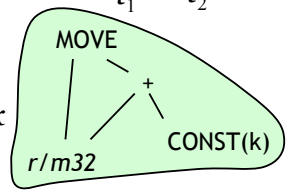


# Statement tiles as rules

$$\frac{e_1 \Rightarrow c_1\,(\text{reg } t_1) \quad e_2 \Rightarrow c_2\,(\text{reg } t_2)}{\text{MOVE}(\text{MEM}(e_1), e_2) \Rightarrow c_1\,;\,c_2\,;\,\textbf{mov}\,[\,t_1\,],\ t_2}$$




$$\frac{e_{1,2} \Rightarrow op_1 \quad (op_1 \text{ is r/m32})}{\text{MOVE}(e_1, e_2 + \text{CONST}(k)) \Rightarrow \textbf{add}\ op_1,\ k}$$




These rules are *not* syntax-directed – need heuristic or dynamic programming to choose