### 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 diagram](image)

**two translations**

#### 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
Compiling function bodies

- Function body:
  \[ S \left[ f(\ldots) : T = e \right] = \text{MOVE}(RV, E \left[ e \right]) \]
  \[ S \left[ f(\ldots) = e \right] = \text{EXP}(E \left[ e \right]) \]

- Variables:
  \[ E \left[ v \right] = \text{TEMP}(t_v) \text{ for locals} \]
  \[ E \left[ v \right] = \text{MEM}(\text{TEMP}(FP) + k) \text{ for args} \]
- Try it out:
  \[ f(x: \text{int}, y: \text{int}) = x + y \]

Abstract assembly for \( f \)

\( f(x: \text{int}, y: \text{int}) = x + y \)

What’s missing here?

Stack frame setup

- Need code to set up stack frame on entry

Function code

\[ f: \text{push bp} \]
\[ \text{mov bp, sp} \]
\[ \text{sub sp, 4}\*l \]
\[ \text{mov t1, [ebp+8]} \]
\[ \text{mov t2, t1} \]
\[ \text{add t2, [ebp+12]} \]
\[ \text{mov eax, t2} \]

\[ f_{\text{epilogue}}: \]
\[ \text{mov sp, bp} \]
\[ \text{pop bp} \]
\[ \text{ret} \]

The Glory of Pentium CISC

\[ f: \text{push ebp} \]
\[ \text{enter 4}\*l, 0 \]
\[ \text{mov ebp, esp} \]
\[ \text{sub esp, 4}\*l \]
\[ \text{mov t1, [ebp+8]} \]
\[ \text{mov t2, t1} \]
\[ \text{add t2, [ebp+12]} \]
\[ \text{mov eax, t2} \]

\[ f_{\text{epilogue}}: \]
\[ \text{mov esp, ebp} \]
\[ \text{leave} \]
\[ \text{pop ebp} \]
\[ \text{ret} \]
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 \( \text{MEM}(\text{FP}+n) \) translated to operand \( [	ext{esp}+(n+k)] \) instead of to \( [\text{ebp}+n] \)
- Advantage: get whole extra register to use when allocating registers (??)

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(...);
  - Iota: x: array[int] = new int[5] (0);

Program Heap

- Program has 4 memory areas: code segment, stack segment, static data, heap
- Two typical memory layouts (OS-dep.):
  - Stack frame setup
  - Sub stack, 4l
  - Add stack, 4l

Object allocation

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

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(...);
  - Iota: x: array[int] = new int[5] (0);

Iota dynamic structures

- Let's consider the following example:

```
a: array[T]
a.length
```

New `T[n]`:

```
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 || g);
a_1 = a_1 + 4;
```

- 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

- High-level source code
- Assembly code
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