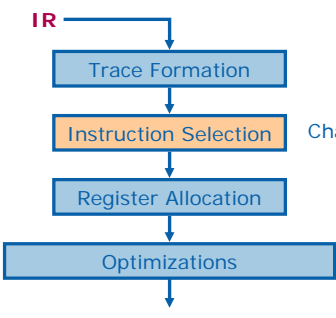


Instruction Selection

CS 471
November 7, 2007



The Back End



Chapter 9

CS 471 - Fall 2007

Current Step: Instruction Selection

Start from an intermediate representation tree:

```

    MEM
    |
    BINOP
   /  \
  PLUS e CONST
         |
         c
    
```

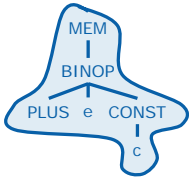
Find the best machine instruction(s) for that tree

`LOAD r1 ← M[fp+c]`

CS 471 - Fall 2007

How?

- Each instruction is represented by a **tree pattern**
- Instruction selection consists of **tiling** the tree with a minimal set of non-overlapping tree patterns



Approaches

- Maximal Munch
- Dynamic Programming

CS 471 - Fall 2007

Example

`x = x + 1;`

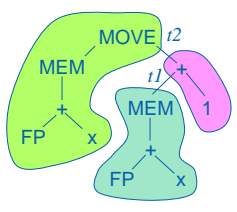
```

    MOVE
   /  \
  MEM  +
 /  \  |
FP  x  MEM  1
      /  \
     FP  x
    
```

CS 471 - Fall 2007

Example

`x = x + 1;`



```

mov t1, [bp+x]
mov t2, t1
add t2, 1
mov [bp+x], t2
    
```

CS 471 - Fall 2007

Tiles

mov t2, t1
add t2, imm8

- Tiles capture compiler's understanding of instruction set
- Each tile: sequence of instructions that update a fresh temporary (may need extra mov's) and associated IR tree
- All outgoing edges are temporaries

6 CS 471 – Fall 2007

Alternate (CISC) tiling

x = x + 1;

add [bp+x], 1

7 CS 471 – Fall 2007

ADD statement tiles

Intel Architecture Manual, Vol 2, 3-17 (online)

add eax, imm32
add r/m32, imm32
add r/m32, imm8
add r/m32, r32
add r32, r/m32

8 CS 471 – Fall 2007

ADD expression tiles

mov t1, t2
add t1, r/m32

9 CS 471 – Fall 2007

Designing tiles

Only add tiles that are useful to compiler
Specific tiles will depend on the architecture
Option: One tile per instruction

- Many instructions will be too hard to use effectively or will offer no advantage

Need tiles for all single-node trees to guarantee that every tree can be tiled, e.g.

mov t1, t2
add t1, t3

10 CS 471 – Fall 2007

Tile Examples

ADD ri ← rj + rk

ADDI ri ← rj + c

LOAD ri ← M[rj+c]

11 CS 471 – Fall 2007

Alternative Tilings

Trees can usually be tiled many ways

Example: `a[i] := x`

```
LOAD r1 ← M[fp + a]    LOAD r1 ← M[fp + a]
ADDI r2 ← r0 + 4        ADDI r2 ← r0 + 4
MUL r2 ← r1 x r2        MUL r2 ← r1 x r2
ADD r1 ← r1 + r2        ADD r1 ← r1 + r2
LOAD r2 ← M[fp + x]     ADDI r2 ← fp + x
STORE M[r1+0] ← r2      MOVEM M[r1] ← M[r2]
```

12

CS 471 – Fall 2007

Challenges

- How do we find the best tiling (efficiently)?
- What does *best* mean anyway?

Optimal tiling – no two adjacent tiles can be combined into a single tile of lower cost

Optimum tiling – all tiles sum to lowest cost

- How do we know the costs?
- Are we missing anything?

13

CS 471 – Fall 2007

Implementation – Maximal Munch

- Start from root node
- Find largest tile covering top node and matching all children
- Invoke recursively on all children of *tile*
- Generate code for this tile (code for children will have been generated already in recursive calls)

- This is a **top-down** tiling algorithm
- Generates instructions in **reverse** order

14

CS 471 – Fall 2007

Maximal Munch in C

```
Static void munchStm(T_stm s) {
  switch(s->kind)
  case T_MOVE:
    if (dst == MEM)
      if (dst->MEM->kind == BINOP)
        /* MOVE(MEM(BINOP(... */
        munchExp(e1); munchExp(e2); emit("STORE");
```

15

CS 471 – Fall 2007

Tile Specifications

Previous approach simple, efficient, but hard-codes tiles and their priorities

Another option: explicitly create data structures representing each tile in instruction set

Tiling performed by a generic tree-matching and code generation procedure

→ A Code Generator Generator!

16

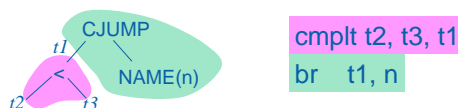
CS 471 – Fall 2007

Matching CJUMP for RISC

`CJUMP(cond, destination)`

Appel: `CJUMP(op, e1, e2, destination)` where *op* is one of `==, !=, <, <=, >, >`

Our CJUMP translates easily to RISC ISAs that have explicit comparison instructions



17

CS 471 – Fall 2007

Condition code ISA

Appel's CJUMP corresponds more directly to Pentium conditional jumps

```
cmp t1, t2
jl n
```

18 CS 471 – Fall 2007

Pentium CJUMP tile

```
cmp t1, t2
jl n
```

How to tile?

19 CS 471 – Fall 2007

Fixed-register instructions

```
mul r/m32
    Sets eax to low 32 bits of eax * operand,
    edx to high 32 bits
```

```
jecxz label
    Jump to label if ecx is zero
```

```
add eax, r/m32
    Add to eax
```

How should we handle this?

20 CS 471 – Fall 2007

Strategies for fixed regs

Use extra mov's and temporaries

```
mov eax, t2
mul t3
mov t1, eax
```

Don't use instruction (jecxz)
Let assembler figure out when to use

21 CS 471 – Fall 2007

Improving instruction selection

- Because greedy, Maximal Munch does not necessarily generate best code
- Always selects largest tile, not necessarily fastest instruction
- May pull nodes up into tiles inappropriately
- Can do better using **dynamic programming** algorithm

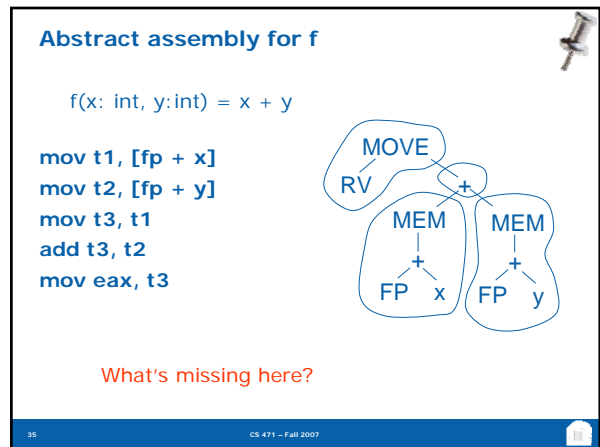
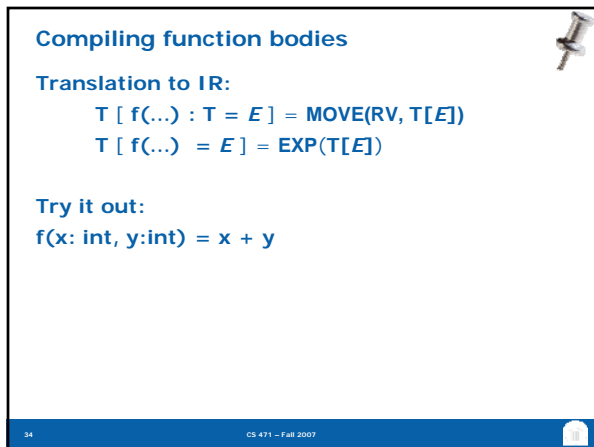
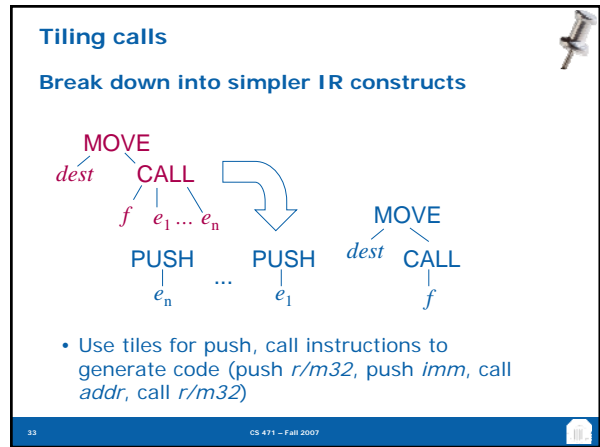
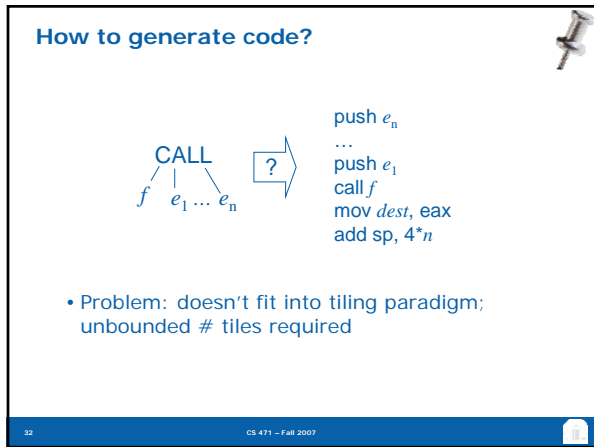
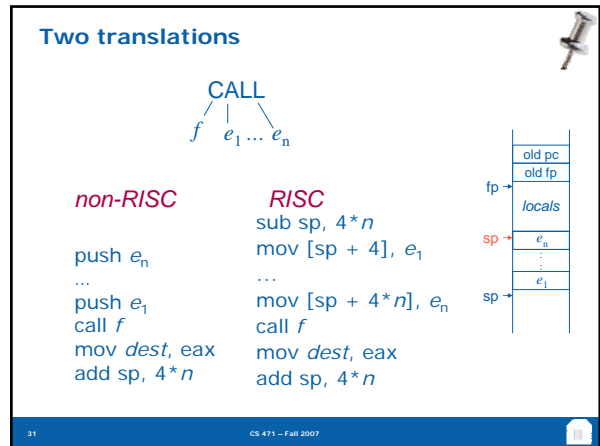
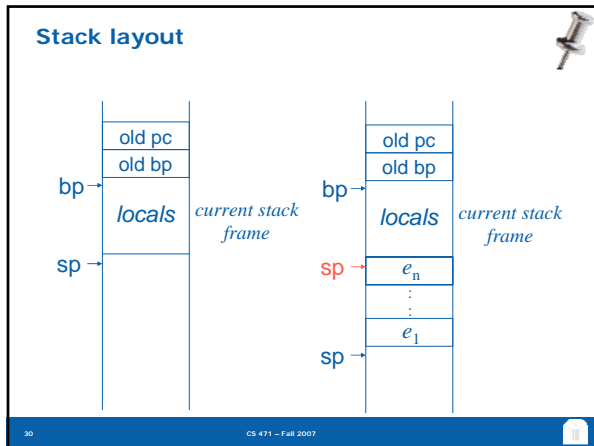
22 CS 471 – Fall 2007

Timing model

- Idea: associate *cost* with each tile (proportional to # cycles to execute)
- Estimate of total execution time is sum of costs of all tiles

Total cost: 5

23 CS 471 – Fall 2007



Stack frame setup

Need code to set up stack frame on entry

36 CS 471 – Fall 2007

Function code

```
f: push bp
   mov bp, sp
   sub sp, 4*I
   mov t1, [fp + x]
   mov t2, [fp + y]
   mov t3, t1
   add t3, t2
   mov eax, t3
   mov sp, bp
   pop bp
   ret
```

function prologue

function epilogue

37 CS 471 – Fall 2007

Compiling returns

Return statement returns immediately from function. Translation:

$$T [\text{return } E;] = \text{SEQ}(\text{MOVE}(\text{RV}, T[E]), \text{JUMP}(\textit{epilogue}))$$

Every function f has epilogue label

```
f: prologue
   body
f_epilogue: epilogue
```

38 CS 471 – Fall 2007

Optimizing away bp

- **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 $FP+n$ translated to operand $sp + (n + k)$ instead of to $bp + n$
- **Advantage:** get whole extra register to use when allocating registers!

39 CS 471 – Fall 2007

Stack frame setup

40 CS 471 – Fall 2007

Caveats

- Get even **faster** (and RISC-core) prologue and epilogue than with enter/leave but:
- Must **save bp register** if we want to use it (like sp, 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 when arrays heap-allocated

41 CS 471 – Fall 2007

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;`

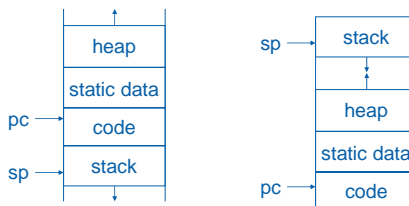
42

CS 471 – Fall 2007

Program Heap

Program has 4 memory areas: code segment, stack segment, static data, heap

Two typical memory layouts (OS-dep.):



43

CS 471 – Fall 2007

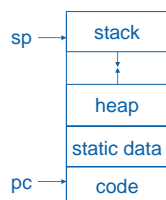
Object allocation

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`

Constants statically allocated in data segment

- string constants
- assembler supports data segment declarations

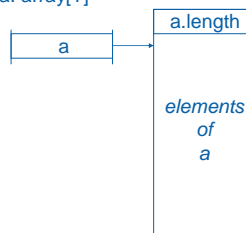


44

CS 471 – Fall 2007

Dynamic structures

`a: array[T]`



`new T[n] = (e)`

```

a = malloc(n * 4 + 4);
MEM(a) = n;
a = a + 4;
for (i = 0; i < n; i++) {
    MEM(a + 4*i) = e[i];
}
    
```

45

CS 471 – Fall 2007

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`

46

CS 471 – Fall 2007

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] => ?
    
```

47

CS 471 – Fall 2007

Result

- Simple way to get working code
- Code is longer than necessary, slower
- Also can allocate temporaries to registers until registers run out (few temporaries on Pentium, more on MIPS, Alpha, lots on Itanium)
- Code generation technique actually used by some compilers when all optimization turned off (-O0)

48

CS 471 – Fall 2007

Summary

- Now: complete code generation technique
- Use **tiling** to perform instruction selection
- Function code generated by gluing **prologue**, **epilogue** onto body
- Dynamic structure allocation handled by relying on heap allocation routines (malloc)
- Allocate temporaries to stack locations to eliminate use of unbounded # of registers

• **Next Time: Register Allocation!**

PA6 due Friday night!

49

CS 471 – Fall 2007