



## DEMO

*How is this done in x86?*

[ifStm.s and whileStm.s]

6 CS 471 – Fall 2007

## Where we are

abstract syntax tree  
*translation functions*  
 intermediate code  
*canonicalization* ←  
 canonical intermediate code  
*code generation*  
 assembly code

7 CS 471 – Fall 2007

## Why canonical form?

**Intermediate code has general tree form**

- easy to generate from AST, but...

**Hard to translate directly to assembly**

- assembly code is a sequence of statements
- Intermediate code has nodes corresponding to assembly statements deep in expression trees

**Canonical form: all statements brought up to top level of tree -- can generate assembly directly**

8 CS 471 – Fall 2007

## Mismatches: Trees vs. Code

- **CJUMP** can jump to two labels (real code “falls through”)
- **ESEQ** nodes within expressions are inconvenient (order of evaluation matters)
- **CALL** nodes within expressions also depend on order (have side effects)
- **CALL** nodes within call nodes cause problems

• **Transform IR to eliminate above cases**

9 CS 471 – Fall 2007

## Canonical form

In canonical form, all SEQ nodes go down right chain:

```

  graph TD
    s1 --- SEQ1[SEQ]
    SEQ1 --- s2
    SEQ1 --- SEQ2[SEQ]
    SEQ2 --- s3
    SEQ2 --- SEQ3[SEQ]
    SEQ3 --- s4
    SEQ3 --- SEQ4[SEQ]
    SEQ4 --- s5
    SEQ4 --- SEQ5[SEQ]
    SEQ5 --- dots[...]
  
```

Function is just one big SEQ containing all statements: SEQ(s1,s2,s3,s4,s5,...)  
 Can translate to assembly more directly

10 CS 471 – Fall 2007

## Helper Routines

**Linearize** – removes ESEQs and moves CALLs to top level

**BasicBlocks** – groups statements into sequences of straight-line code

**TraceSchedule** – orders BBs so that every CJUMP is followed by its false label

(All found in the canon.c file)

11 CS 471 – Fall 2007

### Introducing Temporaries

If  $e_1$  does not *commute* with  $s_1$

- i.e.,  $\{s_1; e_1; e_2\} \neq \{e_1; s_1; e_2\}$

Must save value of  $e_1$  in temporary

12 CS 471 – Fall 2007

### General case

When we move all ESEQ nodes to top, arbitrary expression node looks like:

- ESEQ transformation takes arbitrary expression node, returns list of sub-statements to be executed plus final expression.
- ESEQ node built as shown

13 CS 471 – Fall 2007

### Function Calls, Returns

Tree language permits CALL nodes as subexpressions, e.g.

$\text{BINOP(PLUS, CALL(...), CALL(...))}$   
but ... second CALL will overwrite RV

Solution:  
 $\text{CALL(fun,args)} \rightarrow \text{ESEQ(MOVE(TEMP t, CALL(fun,args)), TEMP t)}$

Then resume ESEQ elimination

14 CS 471 – Fall 2007

### Canonicalization

15 CS 471 – Fall 2007

### Flattening SEQ nodes

All SEQ nodes now top level or children only of SEQ nodes

No ESEQ nodes -- all statement nodes are children of SEQ nodes

All CALL nodes children of EXP nodes or MOVE nodes at top level

Final step: *linearize* SEQ nodes: in-order

16 CS 471 – Fall 2007

### Conditional Jumps

IR is now just a linear list of statements

Still contains CJUMP nodes : two-way branches

Real machines : fall-through branches (e.g. JZ, JNZ)

CJUMP e, t, f  
 ...  
 LABEL(t)  
 if-true code  
 LABEL(f)

evaluate e  
 JZ f  
 if-true code  
 f:

17 CS 471 – Fall 2007

## Basic Blocks

A **basic block** is a sequence of statements that is always entered at its start and always exits at the end:

- starts with a LABEL(n) statement
- ends with a JUMP or CJUMP statement
- contains no other JUMP or CJUMP statement
- contains no interior LABEL that is used as the target for a JUMP or CJUMP from elsewhere

(Learned algorithm for splitting into basic blocks last time...)

18

CS 471 – Fall 2007

## Fixing Conditional Jumps

Reorder basic blocks so that (if possible)

- the “false” direction of two-way jumps goes to the very next block
- JUMPs go to the next block

What if not satisfied?

- For CJUMP add another JUMP immediately after to go to the right basic block

How to find such an ordering of the basic blocks?

19

CS 471 – Fall 2007

## Traces

Idea: order blocks according to a possible *trace*: a sequence of blocks that might (naively) be executed in sequence, never visiting a block more than once

Algorithm:

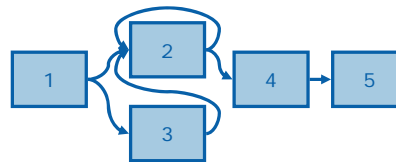
- pick an unmarked block (begin w/ start block)
- run a trace until no more unmarked blocks can be visited, marking each block on arrival
- repeat until no more unmarked blocks

20

CS 471 – Fall 2007

## Example

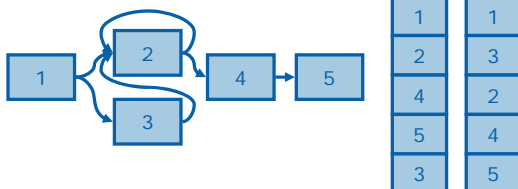
Possible traces?



21

CS 471 – Fall 2007

## Arranging by traces



Advantages?

22

CS 471 – Fall 2007

## Trace Selection

Trace selection is important for code layout  
Code layout is important for performance

Leverages:

- Heuristics
- Profiling information

Impacts performance

- Instruction cache locality
- Prefetching

23

CS 471 – Fall 2007

## Traces



- IR Trees are not linear
- Code is linear
- One path through a tree is a trace
- Trace selection is a very important optimization (often uses profiling information)
  
- Read Chapter 8
  
- Next Time:
  - Instruction Selection

