Administration

• Functions Project, phase 3
  – Translate Functions programs to IR
  – Deadline is next Wednesday

• Midterm
  – Thursday March 1st in class
Find the Basic Blocks

\begin{verbatim}
MOVE(t000 <- %rdi)
MOVE(t001 <- BINOP(MINUS, t000, CONST 1))
CJUMP(LT, t000, CONST 1, _L_3, _L_4)
LABEL _L_3
MOVE(t002 <- CONST 1)
JUMP(NAME(_L_5))
LABEL _L_4
MOVE(t014 <- t000)
MOVE(t013 <- CALL(NAME(__fact), t001))
MOVE(t002 <- BINOP(MUL, t014, t013))
LABEL _L_5
MOVE(%rax <- t002)
LABEL _bail_2
\end{verbatim}
BasicBlocks

```java
private void mkBlocks(List<IRStm> l) {
if (!l.head().instanceof LABEL) //Must start with a LABEL.
    l = List.cons(LABEL(Label.gen()), l);

    for (IRStm stm : l) {
        if (stm.isJump()) {
            currentBlock.add(stm);
            endCurrentBlock();
        } else if (stm instanceof LABEL) {
            startNewBlock((LABEL)stm);
        } else
            currentBlock.add(stm);
    }

    if (currentBlock!=null) //We "fell of the end" without a JUMP.
        doneLabel = Label.generate("DONE");
        currentBlock.add(JUMP(doneLabel));
        endCurrentBlock();
}
```
Three steps

• Convert to linearized, canonical IR
  – no ESEQ nodes, all SEQ nodes clustered at the top => list
  – CALL only in restricted places:
    • EXP(CALL(...))
    • MOVE(tmp, CALL(...))

• Convert linearized IR into basic blocks
  – A basic block:
    • starts with a label
    • no “internal” jumps or labels
    • ends with a jump

• Trace Scheduling:
  – analyze and optimize JUMP and CJUMP nodes.
Trace Scheduling

Now we have a list of basic blocks ... represented by
- List<List<IRStm>>
- a “done” Label

The basic blocks all
- begin with a LABEL
- have no internal jumps or labels
- end with a JUMP or CJUMP

=> The basic blocks can go in any order without changing the meaning!

Q: So what order do we choose?

=> Trace Scheduling = Choose an order for the basic blocks
Terminology: What's a Trace?

Trace: sequence of statements that may be executed consecutively in some execution of the program.

Trace Scheduling:
- computing a covering set of traces:
  - each statement in the program is part of exactly one trace.
public class TraceSchedule {

/**
 * We will be rebuilding the IR code in here, from the basic
 * blocks.
 */
private List<IRStm> program = List.empty();

/**
 * Table to find a bb for a given label. Also, when a block
 * is “covered” it is removed from the map.
 */
private Map<Label, List<IRStm>> table =
    new HashMap<Label, List<IRStm>>();
void trace(Label startAt) {
    List<IRStm> block = table.get(startAt);
    if (block!=null) {
        table.remove(startAt); // only trace each block once!

        //Loop through all statements except the last one:
        for (; !block.tail().isEmpty(); block = block.tail()) {
            IRStm stm = block.head();
            program.add(stm);
        }

        //Process last statement.
        IRStm last = block.head();
        Assert.assertTrue(last.isJump());
        if (last instanceof CJUMP) {
            ... handle CJUMP case ...
        } else {
            ... handle JUMP case ...
        }
    }
}
if (last instanceof CJUMP) {
    CJUMP cjump = (CJUMP) last;
    Label falseTarget = cjump.getFalseLabel();
    Label trueTarget = cjump.getTrueLabel();

    if (table.containsKey(falseTarget)) {
        program.add(cjump);
        trace(falseTarget);
    }
    else if (table.containsKey(trueTarget)) {
        program.add(cjump.flip());
        trace(trueTarget);
    }
    else {
        Label newFalseLabel = Label.gen();
        program.add(cjump.changeFalseLabel(newFalseLabel));
        program.add(LABEL(newFalseLabel));
        program.add(JUMP(falseTarget));
    }
}
### Flipping a CJUMP

```java
public class CJUMP extends IRStm {
    private RelOp relop;
    private IRExp l, r;
    private Label iftrue, iffalse;

    ...

    /**
     * Flip the branches of the CJUMP (without changing its
     * meaning: the condition is also "flipped").
     */
    public CJUMP flip() {
        return new CJUMP(relop.not(), l, r, iffalse, iftrue);
    }
}
```
if (last instanceof CJUMP)
...
else { // Regular jump
    List<Label> targets = last.getJumpTargets();
    Label target = targets.head();
    if (targets.size() != 1) {
        // JUMP with dynamic target should not be dropped!
        program.add(last);
    }
    else { // JUMP(NAME(target))
        if (table.containsKey(target) || target == doneLabel && table.isEmpty()) {
            // Drop this jump
        }
        else
            program.add(last);
    }
}
trace(target);
Review (and a switch of “ends”)

• What have we done so far?
  – A whirlwind trip through a simple compiler

• Front end processing
  – Scanning (regular expressions)
  – Parsing (context-free grammars and LL(1) parsing)
  – Scopes
  – Type checking
  – Runtime memory organization (stack frames)
  – Translation to IR code
  – Canonical IR

• On to the back end!
Done with IR – on to real code!

• This form should be “easy” to convert into Assembly, but...
  – Different processors have different instruction sets.
• How to pick the best (or at least a good) sequence of instructions for some particular processor?
• This problem is called “instruction selection”.
• Chapter 9 explains some of the algorithms used to tackle this problem.
X86_64

- There are references on the course web page.
Tree Patterns

- Idea: assembly instructions correspond to (IR) tree patterns.
- Example:
  - Most machine languages have a “register + offset” form of memory addressing.

```
MEM
  | BINOP
  |   e
  |   CONST
    |   k
```

*e and k are pattern variables.*
Tree patterns – how do we define them?

- Machine instructions can be quirky, but there is often a nice regular subset
  - movq
  - addq
  - jcc
  - jmp
- It is much easier for a compiler to ignore the quirky instructions and only use a small subset
- Using the quirky instructions means writing some more complicated tree patterns
Tree patterns – examples

- **X86_64 addq instruction**
  - `addq $i, %rxx`
  - `addq $i, o(%ryy)`
  - `addq o(%rxx), %ryy`
  - `addq %rxx, o(%ryy)`
  - `addq %rxx, %ryy`
Tree Patterns (Add literal and register)

```
addq $i, %rx
```

```
MOVE

BINOP

PLUS TEMP CONST

TEMP

`tn` and `k` are pattern variables

`tn` has to be the same in both
Tree Patterns (Add literal and register)

\[
\text{addq } $i, \ %rxx \\
\]

e and \( k \) are pattern variables.

\[
\text{BINOP} \\
\text{PLUS} \quad e \quad \text{CONST} \\
\]

\( k \)