Administration

• Functions Project, phase 3
  – Translate Functions programs to IR
  – Deadline was yesterday

• MiniJava Project, phase 1
  – Scanning and Parsing, due Wednesday February 28th

• Midterm Break
  – February 19-23

• Midterm
  – Thursday March 1st in class
A simple X86_64 instruction set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>%rxx, %ryy</td>
</tr>
<tr>
<td>subq</td>
<td>%rxx, %ryy</td>
</tr>
<tr>
<td>imulq</td>
<td>%rxx, %ryy</td>
</tr>
<tr>
<td>movq</td>
<td>$const, %rxx</td>
</tr>
<tr>
<td>movq</td>
<td>%rxx, %ryy</td>
</tr>
<tr>
<td>movq</td>
<td>(%rxx), %ryy</td>
</tr>
<tr>
<td>movq</td>
<td>%rxx, (%ryy)</td>
</tr>
<tr>
<td>cmpq</td>
<td>%rxx, %ryy</td>
</tr>
<tr>
<td>cmovcc</td>
<td>%rxx, %ryy</td>
</tr>
<tr>
<td>jcc</td>
<td>label</td>
</tr>
<tr>
<td>jmp</td>
<td>label</td>
</tr>
<tr>
<td>call</td>
<td>label</td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Selection

Idea: cover the IR tree with a set of disjoint “tiles”. Each tile corresponds to an “instruction pattern”.

![Diagram of instruction selection]
Suppose we expand our X86_64 instruction set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Immediate</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>addq</td>
<td>0(%rxx)</td>
<td>%ryy</td>
</tr>
<tr>
<td>addq</td>
<td>$\text{const}$</td>
<td>%rxx</td>
</tr>
<tr>
<td>movq</td>
<td>0(%rxx)</td>
<td>%ryy</td>
</tr>
<tr>
<td>movq</td>
<td>%rxx</td>
<td>0(%ryy)</td>
</tr>
</tbody>
</table>
Multiple Tilings are Possible

Exercise: Find another valid tiling for this tree.
Optimum Tiling = “best” tiling

• What is the meaning of “best”?
  – fastest execution
  – least number of instructions
  – smallest code size
  – ...

• Must define some “cost” function to define “best”

• Assuming cost is defined (e.g., as estimated execution time)
  – Optimum tiling = Tiling of least cost
Optimum vs. Optimal Tiling

- Optimum = global optimum.
- Optimal = a kind of local optimum.
  - cannot be improved by merging adjacent tiles.
- Can there be more than one optimum tiling?
- Can there be more than one optimal tiling?
- Is an optimum tiling also an optimal tiling?
- Is an optimal tiling also an optimum tiling?
- What’s easier to find: an optimal or optimum tiling?
Optimal tiling: Maximal Munch

• Find largest tile that fits at the root of the tree.
• Repeat the algorithm on each of the remaining subtrees (i.e., the trees you are left with after “removing” the matched tile).
• That’s it!
• Does this indeed produce an optimal tiling?
Optimal tiling: Maximal Munch

• Does this indeed produce an optimal tiling? Yes.

• Proof:
  – Assume that it does not produce an optimal tiling.
  – Then there must be some subtree where the root can be joined with at least one of its children to give a bigger tile. This is impossible since the algorithm always selects the biggest tile possible, so the tiles would have been merged by the algorithm.
Implementing Maximal Munch

• Implement three functions:

/** Emits code as a side effect */
void munchStm( IRStm st) { .... }

/** Emits code as a side effect. Returns the name of the Temp where the result of the IRExp is stored. */
Temp munchExp( IRExp e) { .... }

/** Emits code as a side effect */
void munchData( IRData data) { .... }
Implementing Maximal Munch

- In the book these functions contain manually written “pattern matching code” to recognize tile shapes.
- For example: (page 181 in the book)

```java
void munchMove(MEM dst, Exp src) {
    if (dst.exp instanceof BINOP
        && ((BINOP)dst.exp).oper==BINOP.PLUS
        && ((BINOP)dst.exp).right instanceof CONST) {
        ...
    }
    else if (... more messy instanceof stuff ...)
    {
```
There is a Better Way

- Rather than just pretending to have tree patterns which are implemented in code
- We can explicitly represent the patterns in objects
- This gets a bit hairy
  - kudos to Kris de Volder, who put it together
Step 1

- Implement a representation for IR patterns that can be matched to IR trees.

```java
public abstract class Pat<N> {
    public Matched tryMatch(N toMatch) { ... }
    public int size() { ... }

    public static <N> Pat<N> any() { return new Wildcard<N>(); }

    ... more stuff in here ...
    ... will explain some of it later ...
}
```
Step 2

• Embed the patterns in rules.

```java
private static MuncherRules<IRExp, Temp> em =
    new MuncherRules<IRExp, Temp>();
em.add(new MunchRule<IRExp, Temp>
    (PLUS(_e_, CONST(_i_))) {
        @Override
        protected Temp trigger(Muncher m, Matched c) {
            Temp sum = new Temp();
            m.emit( A_MOV(sum, m.munch(c.get(_e_))) );
            m.emit( A_ADD(sum, c.get(_i_)) );
            return sum;
        }
    });
```

Step 2 (continued)

- Embed the patterns in rules.

```java
private static MuncherRules<IRStm, Void> sm =
    new MuncherRules<IRStm, Void>();

sm.add(new MunchRule<IRStm, Void>
    ( MOVE(TEMP(_t_), _e_) ) {

    @Override
    protected Void trigger(Muncher m, Matched c) {
        m.emit(A_MOV(c.get(_t_),
                        m.munch(c.get(_e_)) ));
        return null;
    }

```
Step 3

• Make the patterns more clever.

\[
\text{PLUS}(\_e\_, \text{CONST}(\_i\_))
\]

• Knows that addition is commutative

```java
public static Pat<IRExp> PLUS(
    Pat<IRExp> l, Pat<IRExp> r) {
    return or(
        BINOP(Op.PLUS, l, r),
        BINOP(Op.PLUS, r, l) );
}
```
Step 4

• Sort the patterns by size
• The MuncherRules automatically get sorted based on their pattern size (so you don’t have to worry about the ordering in which you write/add your rules to the muncher).
Emitting Assembly Code

• We now understand how maximal munch can be implemented with “munching rules”:
• A rule:
  – matches a pattern in the IR tree
  – “largest tile” pattern takes priority
• when triggered the rule:
  – recursively calls the matcher on subtrees
  – emits assembly code for the pattern
    (we haven’t discussed this yet)
• We need a good representation of Assembly code:
  – not target architecture specific
  – must allow assembly code with “fictional temp
    registers”.
  – must have enough information for register allocator to
    analyze register usage
• The representation must be
  – abstract w.r.t. target architecture
  – concrete and detailed w.r.t register usage and flow of
    control.
Assembly Code – a list of instructions

- What do we care about?
  - Labels
  - Branches
  - Registers
    - Which ones get used (read)
    - Which ones get changed (written)

- What don’t we care about?
  - Instructions (\texttt{sub} vs. \texttt{add} vs. \texttt{div})
Assembly Code – Instructions

• For each instruction
  – Where might it branch to?
  – What registers does it read?
    • “use” set
  – What registers does it write?
    • “def” set

public class Instr {
  abstract public List<Label> jumps();
  abstract public List<Temp> def();
  abstract public List<Temp> use();
}
• Labels are special as jump targets.

```java
public class A_LABEL extends Instr {
    private Label label;
    public List<Temp> use() {
        return List.empty();
    }
    public List<Temp> def() {
        return List.empty();
    }
    public List<Label> jumps() {
        return null;
    }
}
```
Assembly Code – Moves

• Moves are special, too. Stay tuned ...

public class A_MOVE extends Instr {
    private Temp src, dst;
    public List<Temp> use() {
        return list(src);
    }
    public List<Temp> def() {
        return list(dst);
    }
    public List<Label> jumps() {
        return null;
    }
}

Instruction Selection (Chapter 9)
Assembly Code – Jumps

- Jumps are special – represent control flow

```java
public class A_JMP extends Instr {
    private Label target;
    public List<Temp> use() {
        return List.empty();
    }
    public List<Temp> def() {
        return List.empty();
    }
    public List<Label> jumps() {
        return list(target);
    }
}
```
Assembly Code – Jumps

- Jumps are special – represent control flow

    public class A_CJMP extends Instr {
        private Label thn, els;
        public List<Temp> use() {
            return List.empty();
        }
        public List<Temp> def() {
            return List.empty();
        }
        public List<Label> jumps() {
            return list(thn, els);
        }
    }
For every other instruction

```java
public class A_OPER extends Instr {
    public List<Temp> dst, src;
    public List<Label> jump;
    public List<Temp> def() {
        return dst;
    }
    public List<Temp> use() {
        return src;
    }
    public List<Label> jumps() {
        return jump;
    }
}
```
public class A_OPER extends Instr {
    public A_OPER(a, d, s, j) {
        dst = d;
        src = s;
        jump = j;
    }
    public A_OPER(a, d, s) {
        this(a, d, s, null);
    }
}
Examples

Instr A_ADD(Temp reg, int i) {
    return new A_OPER(
        "addq $\"+i\"`, `d0”,
        list(reg),
        list(reg));
}

Examples

Instr A_ADD(Temp dst, Temp src) {
    return new A_OPER(
        "addq `s0, `d0",
        list(dst),
        list(src, dst));
}
Examples

Instr A_MOVE_TO_MEM

(Temp ptr, Temp src) {
    return new A_OPER(
        "movq `s1, (`s0)",
        List.empty(),
        list(ptr, src));
}
RISC machines

- 32 registers
- one class of “general purpose” integer/pointer register
- arithmetic only between registers
- 3-address instructions: \( r1 \leftarrow r2 \text{ OP } r3 \)
- load/store only with \( M[\text{reg}+\text{const}] \)
- every instruction same size (32 bits)
- one result or effect per instruction
CISC machines

- few registers (e.g., 6 or 8)
- multiple register classes, special purpose registers
- arithmetic also on memory
- 2-address instructions:
  - r1 <- r1 OP r2
  - eax <- eax + r2
- complex addressing modes
- variable instruction size
- some instructions have multiple effects/results (e.g., “push”, “leave”)
X86_64 is a CISC machine (sort of)

• 2-address instructions
  – Allocate a new temp, copy, operate

• arithmetic on memory
  \[
  \text{addq } 8(\%\text{rbp}), \%\text{rcx} \quad \#\%\text{rcx} \leftarrow \%\text{rcx} + M[\%\text{rbp}+8] \\
  \text{addq } \%\text{rcx}, 8(\%\text{rbp}) \quad \#M[\%\text{rbp}+8] \leftarrow %\text{rcx}+M[\%\text{rbp}+8]
  \]

• complex addressing modes
  – General form for a mem operand is
    • \text{offset}(r1, r2, \text{scale})
    • where scale is one of: 1, 2, 4, 8.

• some instructions have multiple effects/results
  (e.g., “push”, “leave”)