X86-64 Instruction Selection for Fab

- Assume we have permanently allocated each IR Name and Temp to a register, a stack slot, or (for free variable names) a closure slot.
- So when generating code for an IR operation, need to be able to cope with either register or memory operand in each position.

X86 instructions are much more constrained:

- `mov s, d`  
  - `d` can be reg or mem; `s` can be reg, imm, or mem (but not if `d` is)
- `add s, d`  
  - `d` can be reg or mem; `s` can be reg, imm, or mem (but not if `d` is)
- `sub s, d`  
  - `d` must be reg; `s` can be reg, imm, or mem
- `imul s, d`  
  - `d` must be reg; `s` can be reg, imm, or mem
- `idiv s`  
  - `%rax := %rdx:%rax div s` and `%rdx := %rdx:%rax mod s`  
  - `s` must be reg or mem
- `cmp s1, s2`  
  - Test sense is “backwards”; `s1` can be reg, imm, or mem; `s2` can be reg or mem (but not if `s1` is)

Can use these utility functions:

```c
X86.Operand gen_source_operand (IR.Operand rand, int size, boolean mem_ok, boolean imm_ok, X86.Reg temp)
X86.Operand gen_target_operand (IR.Operand rand, int size, X86.Reg temp)
```

Register Allocation for Expressions

Choice of evaluation order when linearizing expression trees can affect number of registers needed.

Example: Assume a RISC-like load-store instruction set. If we compute left child first, need 4 regs, but doing right child first needs only 3.

```
(a+b) - ((c+d) - (e+f)) -
\[ \text{load } a, r1 \]
\[ \text{load } c, r1 \]
\[ \text{add } r1, r2, r1 \]
\[ \text{add } r1, r2, r1 \]
\[ \text{add } r2, r3, r2 \]
\[ \text{add } r2, r3, r2 \]
\[ \text{add } r3, r4, r3 \]
\[ \text{add } r3, r4, r3 \]
\[ \text{load } e, r3 \]
\[ \text{load } f, r4 \]
\[ \text{add } r4, r5, r4 \]
\[ \text{add } r4, r5, r4 \]
```

Must be smarter in this case, or dumber in the regular case.

Hint: Use `%r10` and `%r11` as temporaries within IR instructions.

- There are many possible improvements, especially when dealing with constants or commutative operators.
MINIMIZING REGISTERS NEEDED TO EVALUATE EXPRESSION TREES

Key idea (Sethi & Ullman): At each node, first evaluate subtree requiring largest number of registers to evaluate. Can then save result of this evaluation in a register while doing other subtree.

1. Label each node with minimum number of registers needed to evaluate subtree.
   
   $$\text{risc_label}(t) \begin{cases} \text{if isLeaf}(t) \text{ then} & t->label = 1 \quad (\text{depends on machine architecture}) \\
   \text{else} & \text{label}(t->left) \quad a^1 \\
   & \text{label}(t->right) \quad b^1 \\
   \text{if (t->left->label == t->right->label)} & \text{then} \quad c^1 \\
   \text{else} & \text{max(t->left->label, t->right->label)} \quad f^1 \end{cases}$$

2. Use labels to guide order of code emission; emit code for higher-numbered subtree first.

OTHER ISSUES IN TREE EVALUATION ORDER

- Some machines allow one operand to be a complex expression, while the other must be a register, which also holds the result (“accumulator” style):
  
  ```
  add 37, r0 ; r0 <- r0 + 37
  add [b], r1 ; r1 <- r1 + *b
  sub r0, r1 ; r1 <- r1 - r0
  ```

  These machines have different Sethi-Ullman numbering, e.g., right leaves might require no temporary registers at all.

- Can use associativity to make trees “less bushy,” e.g.
  
  ```
  +^3 ^2
  / \ / \ \\
  / \ / \\
  / \ / \\
  a^1 b^1 c^1 d^1 f^1
  ```

BASIC BLOCKS

- Extend analysis of register use to program units larger than expressions but still completely analyzable at compile time.
- **Basic Block** = sequence of instructions with single entry & exit.
- If first instruction of BB is executed, so is remainder of block (in order).
- To calculate basic blocks:
  
  (1) Determine BB leaders (→):
  
  (a) First statement in routine
  (b) Target of any jump (conditional or unconditional).
  (c) Statement following any jump.
  (What about subroutine calls?)
  (2) Basic block extends from leader to (but not including) next leader (or end of routine).

BASIC BLOCK EXAMPLE

```
prod := 0;
i := 1;
while i <= 20 do
  prod := prod + a[i] * b[i];
i := i + 1
end
```

```
→ 1. prod := 0
2. i := 1
→ 3. if i > 20 goto 14
4. t1 := i * 4
5. t2 := addr a
6. t3 := *(t2+t1)
7. t4 := i * 4
8. t5 := addr b
9. t6 := *(t5+t4)
10. t7 := t3 * t6
11. prod := prod + t7
12. i := i + 1
13. goto 3
→ 14. ---
```
BB Code Generation using Liveness

Can combine code generation with “greedy” register allocation: bring each variable into a register when first needed, and leave it there as long as it’s needed (if possible).

Maintain register descriptors saying which variable is in each register, and address descriptors saying where (in memory and/or a register) each variable is.

For each IR instruction \( x := y \text{ op } z \):
1. If \( y \) isn’t in a register, load it into a free one, updating descriptors.
2. Similarly for \( z \).
3. If \( y \) and/or \( z \) are no longer live following this instruction, mark their registers as free.
4. Choose a free register for \( x \), updating descriptors.
5. Generate instruction \( \text{op } r_y,r_z,r_x \).

For the special case \( x := y \), load \( y \) into a register, if necessary, and then mark that register as holding \( x \) too.

\[ d := (a-b) + (a-c) + (a-c) \]

IR: \[ t := a - b \]
\[ u := a - c \]
\[ v := t + u \]
\[ d := v + u \]

Descriptors after Instructions

IR Statement | Code | Regs | Addr
--- | --- | --- | ---
\( t := a - b \) | \( \text{ld } a,r0 \) | \( r0:a \) | \( a:r0,\text{mem} \)
\( \text{ld } b,r1 \) | \( r1:t \) | \( b,c:\text{mem} \)
\( \text{sub } r0,r1,r1 \) | \( t:r1 \) | 
\( u := a - c \) | \( \text{ld } c,r2 \) | \( r0:u \) | \( a,b,c:\text{mem} \)
\( \text{sub } r0,r2,r0 \) | \( r1:t \) | \( u:r0 \)
\( v := t + u \) | \( \text{add } r1,r0,r1 \) | \( r0:v \) | \( u:r0 \)
\( v:r1 \)
\( d := v + u \) | \( \text{add } r0,r1,r0 \) | \( r0:d \) | \( a,b,c:\text{mem} \)
\( \text{st } r0,d \) | \( r1:v \) | \( d:r0,\text{mem} \)

BB Code Generation using Liveness (more)

- Must now be careful not to free a register unless none of its associated variables is live.
- Registers behave like a cache for memory locations.
- Nasty problems for source-level debuggers and dump utilities – where is that variable?!

Example (assuming risc-like instruction set)...

Control-flow Graphs

To assign registers on a per-procedure basis, need to perform liveness analysis on entire procedure, not just basic blocks.

To analyze the properties of entire procedures with multiple basic blocks, we use a control-flow graph.

In simplest form, control flow graph has one node per statement, and an edge from \( n_1 \) to \( n_2 \) if control can ever flow directly from statement 1 to statement 2.

We write \( \text{pred}[n] \) for the set of predecessors of node \( n \), and \( \text{succ}[n] \) for the set of successors.

(In practice, usually build control-flow graphs where each node is a basic block, rather than a single statement.)

Example....
LIVENESS ANALYSIS USING DATAFLOW ANALYSIS

Working from the future to the past, we can determine the edges over which each variable is live.

In the example:

- $b$ is live on $2 \rightarrow 3$ and on $3 \rightarrow 4$.
- $a$ is live from on $1 \rightarrow 2$, on $4 \rightarrow 5$, and on $5 \rightarrow 2$ (but not on $2 \rightarrow 3 \rightarrow 4$).
- $c$ is live throughout (including on entry $\rightarrow 1$).

Can see that two registers suffice to hold $a, b, c$.

DATAFLOW EQUATIONS

We can do liveness analysis (and many other analyses) via dataflow analysis.

A node **defines** a variable if its corresponding statement assigns to it.

A node **uses** a variable if its corresponding statement mentions that variable in an expression (e.g., on the rhs of assignment).

For any variable $v$, define

- $\text{def}[v] = \text{set of graph nodes that define } v$
- $\text{use}[v] = \text{set of graph nodes that use } v$

Similarly, for any node $n$, define

- $\text{def}[n] = \text{set of variables defined by node } n$;
- $\text{use}[v] = \text{set of variables used by node } n$.

SETTING UP EQUATIONS

A variable is **live** on an edge if there is a directed path from that edge to a use of the variable that does not go through any def.

A variable is **live-in** at a node if it is live on any in-edge of that node; it is **live-out** if it is live on any out-edge.

Then the following equations hold:

\[
\text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n])
\]

\[
\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]
\]

We want the **least fixed point** of these equations: the smallest $\text{in}$ and $\text{out}$ sets such that the equations hold.
**SOLVING FIXED-POINT EQUATIONS**

We can find this solution by iteration:

- Start with empty sets
- Use equations to add variables to sets, one node at a time.
- Repeat until sets don’t change any more.

Adding additional variables to the sets is safe, as long as the sets still obey the equations, but inaccurately suggests that more live variables exist than actually do.

Again, in practice we normally work on a CFG in which each node is an entire basic block. It is easy to compute the *in* and *out* sets of each block by a simple backward pass over its instructions.

---

**EXAMPLE SOLUTION**

For correctness, order in which we take nodes doesn’t matter, but it turns out to be fastest to take them in roughly reverse order:

<table>
<thead>
<tr>
<th>node</th>
<th>use</th>
<th>def</th>
<th>1st in</th>
<th>2nd in</th>
<th>3rd in</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>c</td>
<td>def</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td></td>
<td>c</td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>a</td>
<td>ac</td>
<td>bc</td>
<td>ac</td>
</tr>
<tr>
<td>3</td>
<td>bc</td>
<td>c</td>
<td>bc</td>
<td>bc</td>
<td>bc</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>b</td>
<td>bc</td>
<td>ac</td>
<td>bc</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td>ac</td>
<td>c</td>
<td>ac</td>
</tr>
</tbody>
</table>

Implementation issues:

- Algorithm always terminates, because each iteration must enlarge at least one set, but sets are limited in size (by total number of variables).
- Time complexity is \(O(N^4)\) worst-case, but between \(O(N)\) and \(O(N^2)\) in practice.

---

**STATIC VS. DYNAMIC LIVENESS**

Consider the following graph:

```
1 V
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b*b</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
2 V
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c = a+b</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
3 V
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c &gt;= b ?</td>
</tr>
<tr>
<td>---------</td>
</tr>
</tbody>
</table>

\/
4 V 5 V
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>return a</td>
</tr>
<tr>
<td>---------</td>
</tr>
</tbody>
</table>

\/
```

Is a live-out at node 2? It depends on whether control flow ever reaches node 4.

---

**STATIC LIVENESS (CONTINUED)**

A smart compiler could answer no.

A smarter compiler could answer similar questions about more complicated programs.

But no compiler can ever *always* answer such questions correctly. This is a consequence of the *uncomputability* of the Halting Problem.

So we must be content with static liveness, which talks about paths of control-flow edges, and is just a *conservative* approximation of dynamic liveness, which talks about actual execution paths.
**Halting Problem**

**Theorem** There is no program $H$ that takes an input any program $P$ and input $X$, and (without infinite-looping) returns true if $P(X)$ halts and false if $P(X)$ infinite-loops.

**Proof** Suppose there were such an $H$. From it, construct the function $F(Y) = \text{if } H(Y,Y) \text{ then (while true do ())) else 1}$

Now consider $F(F)$.

- If $F(F)$ halts, then, by the definition of $H$, $H(F,F)$ is true, so the then clause executes, so $F(F)$ does not halt.
- But, if $F(F)$ loops forever, then $H(F,F)$ is false, so the else clause is taken, so $F(F)$ halts.

Hence $F(F)$ halts if any only if it doesn’t halt.

Since we’ve reached a contradiction, the initial assumption is wrong: there can be no such $H$.

**Reachability Problem**

**Corollary** No program $H'(P, X, L)$ can tell, for any program $P$, input $X$, and label $L$ within $P$, whether $L$ is ever reached on an execution of $P$ on $X$.

**Proof** If we had $H'$, we could construct $H$. Consider a program transformation $T$ that, from any program $P$ constructs a new program by putting a label $L$ at the end of the program, and changing every $\text{halt}$ to $\text{goto } L$. Then $H(P, X) = H'(T(P), X, L)$.

**Register Interference Graphs**

Mixing instruction selection and register allocation gets confusing; need a more systematic way to look at the problem.

- Initially generate code assuming an infinite number of “logical” registers; calculate live ranges. E.g., for previous example:

<table>
<thead>
<tr>
<th>Live after instr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld a,t0</td>
</tr>
<tr>
<td>; a:t0 t0</td>
</tr>
<tr>
<td>ld b,t1</td>
</tr>
<tr>
<td>; b:t1 t0 t1</td>
</tr>
<tr>
<td>sub t0,t1,t2</td>
</tr>
<tr>
<td>; t:t2 t0 t2</td>
</tr>
<tr>
<td>ld c,t3</td>
</tr>
<tr>
<td>; c:t3 t0 t2 t3</td>
</tr>
<tr>
<td>sub t0,t3,t4</td>
</tr>
<tr>
<td>; u:t4 t2 t4</td>
</tr>
<tr>
<td>add t2,t4,t5</td>
</tr>
<tr>
<td>; v:t5 t4 t5</td>
</tr>
<tr>
<td>add t5,t4,t6</td>
</tr>
<tr>
<td>; d:t6 t6</td>
</tr>
<tr>
<td>st t6,d</td>
</tr>
</tbody>
</table>

- Build a register interference graph, which has
  - a node for each logical register.
  - an edge between two nodes if the corresponding registers are simultaneously live.

**Coloring Interference Graphs**

Interference Graph Example:

```
          ____________
          /            \
 t0 ----- t1 t2
          | ______________//
          | / ______/
          |/ /
 t3 t4------ t5 t6
```

A coloring of a graph is an assignment of colors to nodes such that no two connected nodes have the same color. (Like coloring a map, where nodes=countries and edges connect countries with common border.)

Suppose we have $k$ physical registers available. Then aim is to color interference graph with $k$ or fewer colors. This implies we can allocate logical registers to physical registers without spilling.

In general case, determining whether a graph can be $k$-colored is hard (N.P. Complete, and hence probably exponential).

But a simple heuristic will usually find a $k$-coloring if there is one.
Graph Coloring Heuristic

1. Choose a node with fewer than $k$ neighbors.

2. Remove that node. Note that if we can color the resulting graph with $k$ colors, we can also color the original graph, by giving the deleted node a color different from all its neighbors.

3. Repeat until either
   - there are no nodes with fewer than $k$ neighbors, in which case we must spill; or
   - the graph is gone, in which case we can color the original graph by adding the deleted nodes back in one at a time and coloring them.

In our example, heuristic finds a 3-coloring. There cannot be a 2-coloring (why not?).

Each “color” corresponds to a physical register, so 3 registers will do for this example.