Code Generation

- Project 6
  - Canonicalizing Tree objects
  - Basic blocks and program traces
  - Instruction selection
- Translation of Expression Trees
  - Sethi-Ullman numbering (for register usage)
  - Interaction between instruction scheduling and register allocation
Simplified Code Generation

• Our approach
  – Keep all variables in memory
  – Locality of temporary register usage is 1 instruction
  – Generate SPIM code
  – Use canonicalization codes from Appel text
Canonical Tree Objects

- **IDEA:** to correct some of the mismatches between the intermediate representation (IR) and actual machine assembly instructions
  - To make code generation easier by standardizing the Tree objects somewhat
  - Use tree rewriting (a form of code transformation)

- **Steps:**
  - Apply tree rewrites
  - Find basic blocks
  - Organize basic blocks into traces
Canonical Tree Objects

• Contain no SEQ or ESEQ
• How eliminate these?
  – Have to lift them up in the tree through identities in Figure 8.1 (Appel, p 184)
  – Important to know: if two expressions or statements can commute with no effect on the computation
    • Otherwise we may need new temporary locations to store intermediate results to get canonical trees
Canonical Transformations

\[ \text{ESEQ}(s_1, \text{ESEQ}(s_2, e)) = \text{ESEQ}(\text{SEQ}(s_1, s_2), e) \]

\[ \text{BINOP}(\text{op}, \text{ESEQ}(s, e_1), e_2) = \text{ESEQ}(s, \text{BINOP}(\text{op}, e_1, e_2)) \]
Canonical Transformations

BINOP

op    e1    ESEQ

s    e2

ESEQ

MOVE             ESEQ

TEMP    e1

s    BINOP

op    TEMP    e2

t

BINOP(op, e1, ESEQ(s,e2)) =
ESEQ( MOVE( TEMP(t), e1),
     ESEQ(s, BINOP( op,TEMP(t), e2)))

where t is a new temporary location

PROBLEM: s may have side effects that affect value of e1.
SOLUTION: use a temporary to store value of e1.
Tree Rewriting

• Code provided by Appel does these Tree transformations

• Eventually get a SEQ of Tree statements which can be considered a list of statements
Basic Blocks

• Single-entry, single-exit sequences of code
• Used in optimization and as basic element of code generation algorithms
• Appel’s basic blocks
  – Always entered at the beginning and exited at the end
  – Last statement is a JUMP or CJUMP
  – Contains no other LABELs, JUMPs or CJUMPs
Basic Block Construction

- Given sequence of intermediate code statements (*Canon.BasicBlocks*)
  - Scan from beginning to end
  - If find LABEL, start a new basic block
  - If find JUMP or CJUMP end current basic block and start new basic block with next instruction
  - After finish code scan, add JUMP to next block’s LABEL to any block not ended by a JUMP or CJUMP
  - If any block is missing a beginning LABEL, create one for it
Control Flow Graph

- Body of each function is divided into basic blocks
- *Control flow graph* whose nodes are basic blocks and edges are jumps between them
  - Used to approximate possible flow of control through program
  - Analyzed for info allowing machine independent optimizations
  - Formed with blocks in original sequential order of code
Traces

• Can arrange basic blocks in any order and get same program execution
• So we can choose an ordering so that each CJUMP is following by its false label
• Trace - sequence of statements that can be consecutively executed (can include conditional branches)
• Program is a set of traces (see Algorithm 8.2, Appel p 191)
  – Can flatten set of traces back into a linear list of stmts
Basic Blocks

• Used for local register allocation
  – Small set of registers saved for local computation
  – Algorithms to choose which results to save locally in registers
  – Other registers used for quantities needed across region of control flow graph

• Used for simple instruction scheduling; more complex algorithms use parts of a trace to do instruction scheduling
Example, (ASU p 529)

SOURCE CODE

```plaintext
{ prod := 0;
  j := 1;
  do {
    prod := prod + a[j] * b[j];
    j := j+1;
  }
  while j <= 20
}
```

3 ADDRESS CODE

```plaintext
prod := 0;
j := 1;
t1 := 4 * j; (3)
t2 := a[t1];
t3 := 4 * j;
t4 := b[t3];
t5 := t2 * t4;
t6 := prod + t5;
prod := t6;
t7 := j + 1;
j := t7;
if j <= 20 goto (3)
```
Example

3 ADDRESS CODE
(1) prod := 0;
(2) j := 1;
(3) t1 := 4 * j;
(4) t2 := a[t1];
(5) t3 := 4 * j;
(6) t4 := b[t3];
(7) t5 := t2 * t4;
(8) t6 := prod + t5;
(9) prod := t6;
(10) t7 := j + 1;
(11) j := t7;
(12) if j <= 20 goto (3)

Control Flow Graph

B1
(1) prod := 0;
(2) j := 1;

B2
(3) t1 := 4 * j;
(4) t2 := a[t1];
(5) t3 := 4 * j;
(6) t4 := b[t3];
(7) t5 := t2 * t4;
(8) t6 := prod + t5;
(9) prod := t6;
(10) t7 := j + 1;
(11) j := t7;
(12) if j <= 20 goto (3)
Local Register Allocation

• Within a basic block
• Simplest basic block represents a complex expression computation
  – Important to know how to most efficiently translate such expression trees
  – Measure efficiency in number of instructions
  – Try to keep intermediate results in registers to avoid delays in load/store to memory
Code Generation Example

Here, 2nd sequence of instructions is more efficient!
Sethi-Ullman Numbering

• A way of estimating the number of registers needed to evaluate an expression tree (MinRegs)
• Can prove code generated is *optimal* in sense it uses least number of registers
• Can also reorder intermediate code (that is, rewrite the trees) so that a better translation is possible
Sethi-Ullman Algorithm

/*** assume reg to memory instructions are possible and ***/
/*** can’t use same destination register as an operand***/
Visit nodes in postorder traversal of expression tree
  if n is a leaf then
    { if n is leftmost child of its parent then label(n):= 1
     else label(n) := 0;
    }
  else
    /***/allow for reorganization of order of subexpr computation***/
    { let n1, n2,…,nk be the children of n in order of highest to lowest label;
      label(n) := max (label(nj) + j - 1) for (1<=j<=k)
    }
Example 1

• For binary interior node n, label(n) will be either max(l₁,l₂) if its child node labels are l₁ != l₂, or l₁+1 if l₁==l₂

\[ \text{load a, R1} \]
\[ \text{add b, R1} \]
\[ \text{load c, R2} \]
\[ \text{add d, R2} \]
\[ \text{add e, R2} \]
\[ \text{add R1,R2,R2} \]

\( (a+b) -(e-(c+d)) \)

so this expression needs only 2 registers to evaluate
Example 2

How many registers does this expression need to be evaluated \((a-b)+(g*h)) +((c+d)+(e*f))\)?
SPIM version of Sethi-Ullman

• For SPIM codes the invariants assumed previously aren’t true
  – Only register to register instructions are allowed
  – Can use same destination register as operand

• Q: Does the Sethi-Ullman algorithm still work here? If so, why; if not, why not?
What is the minimum number of registers to code this in SPIM?
Instruction Scheduling

• *Delayed load* architecture requires that destination of load not be accessed by some number of clock cycles, although unrelated instructions can execute

• This limitation on *instruction scheduling* (or ordering) interacts with register allocation
  – Allocation and scheduling are interdependent

• Delayed load scheduling (DLS) tries to move loads as early as possible in the schedule
DLS Scheduling

T. Proebsting, C. Fischer, “Linear-time Optimal Code Scheduling for Delayed-Load Architectures, PLDI’91

• Overview
  – If have instruction sequence with R registers, L loads, and (L-1) operations, then
    • Do R loads,
    • Followed by an alternating sequence of L-R pairs,
    • Followed by the remaining R operations
    • Uses Sethi-Ullman numbering
  – If can add registers, may be able to eliminate delays
Example

Three schedules for this expression tree. Last is shortest at cost of another register!

<table>
<thead>
<tr>
<th>Cycle</th>
<th>SU(3)</th>
<th>Canonical(3)</th>
<th>Canonical(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>load m3, r1</td>
<td>load m3, r1</td>
<td>load m3, r1</td>
</tr>
<tr>
<td>2.</td>
<td>load m4, r2</td>
<td>load m4, r2</td>
<td>load m4, r2</td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td>load m5, r3</td>
<td>load m5, r3</td>
</tr>
<tr>
<td>4.</td>
<td>add r1, r2, r2</td>
<td>add r1, r2, r2</td>
<td>load m6, r4</td>
</tr>
<tr>
<td>5.</td>
<td>load m5, r1</td>
<td>load m6, r1</td>
<td>add r1, r2, r2</td>
</tr>
<tr>
<td>6.</td>
<td>load m6, r3</td>
<td></td>
<td>load m1, r1</td>
</tr>
<tr>
<td>7.</td>
<td></td>
<td>add r3, r1, r1</td>
<td>add r3, r4, r4</td>
</tr>
<tr>
<td>8.</td>
<td>add r1, r3, r3</td>
<td>load m1, r3</td>
<td>load m2, r3</td>
</tr>
<tr>
<td>9.</td>
<td>add r2, r3, r3</td>
<td>add r2, r1, r1</td>
<td>add r2, r4, r4</td>
</tr>
<tr>
<td>10.</td>
<td>load m1, r1</td>
<td>load m2, r2</td>
<td>add r1, r3, r3</td>
</tr>
<tr>
<td>11.</td>
<td>load m2, r2</td>
<td></td>
<td>add r4, r3, r4</td>
</tr>
<tr>
<td>12.</td>
<td></td>
<td>add r3, r2, r2</td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>add r1, r2, r2</td>
<td>add r1, r2, r2</td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>add r3, r2, r2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BEST Schedule
Problems

• By moving Loads up in code schedule may increase length of time values need to be in registers and thus increase *register pressure* at arbitrary program points
  
  – *Register pressure* - number of times that MinReg registers will be *live* (their values still needed) in Sethi-Ullman evaluation order
Example

if have only 2 registers, r1, r2 will be live 3 times in left subtree

Sethi-Ullman numbers (register pressure)