Agenda

• Code Generation
• A Simple CodeGen Algorithm
• A getreg Example
• CodeGen with Dynamic Programming
• Dynamic Programming (DP)
• Applying DP to CodeGen
• CodeGen by Pattern Matching
• Best Tiling
• Maximal Much Algorithm
• Array Indexing
• Call Return
Code Generation (CodeGen)

IR code -> Code Generator -> target code

- Goal: Generating correct and high-quality target code
- Main Issue: Instruction selection: Mapping from IR code to target code; that mapping is not unique!
- Factors that affect selection:
  - variety of instructions
  - variety of address modes
  - Availability of resources at that point (free registers)
An Example

Three-address code:  Intermediate representation, saves t1:

\[
t_1 := \text{fp}[-12] \\
\text{fp}[8] := t_1
\]

\[
\text{(MOVE (MEM (BINOP + (NAME fp) 8)))} \\
\text{(MEM (BINOP - (NAME fp) 12)))}
\]

Sparc-like: destination on the RHS; registers ID’ed with %

# Extreme RISC:

\[
\text{add } \%\text{fp},-12,\%\text{r3} \\
\text{ld } [\%\text{r3}], \%\text{r5} \\
\text{add } \%\text{fp},8,\%\text{r4} \\
\text{st } \%\text{r5},[\%\text{r4}]
\]

# Moderate RISC:

\[
\text{ld } [\%\text{fp}-12],\%\text{r5} \\
\text{st } \%\text{r5},[\%\text{fp}+8]
\]

# CISC, allowing memory-to-memory:

\[
\text{move } [\%\text{fp}-12],[\%\text{fp}+8]
\]
CodeGen Algorithms

• **Simple** CodeGen: Map each IR statement independently onto a fixed set of instructions
  – Simple, can be automated (i.e. code generators exist)
  – Not flexible, code quality may be poor

• CodeGen by *Pattern Matching*: Map a group of IR statements (e.g. a subtree) as a unit
  – Allow local optimization

• CodeGen via *Dynamic Programming*: Associate a cost with each IR statement; perform global optimization to find best target code for IR program
  – Most expensive, but cost is acceptable in general
A Simple CodeGen Algorithm

• Input: A sequence of three-address statements from a basic block (BB)
• Approach: Generate code for each statement independently
• Internal Structure:
  – **Register descriptor**: Tracks what is currently in each register. Multiple variables may be available in the same register: \( x := y \)
  – Note similarity to **score board**: HW register track
  – **Address descriptor**: Keeps track of location (L) of variables, values. A location for a value \( x \) is denoted \( L_x \), which could be a register (R), a stack location (S), or a memory address (M)
  – Function **getreg( s )**: Returns a location for storing the result of a three-address statement \( s \)
A getreg Example

getreg( "x := y op z" )

- if Ly is a register, let Lx = Ly, i.e. re-use same register; otherwise:
- if Lz is a register and op commutative, similarly let Lx = Lz; i.e. NOT for divide! otherwise:
- if there is an empty register R, let Lx = R; else:
- if op requires a register, find an occupied register R, store its value in memory at M, let Lx = R, and remember M; otherwise
- Lx = the memory location of x; will be slow; may be final destination
A Simple CodeGen Algorithm, Cont’d

\[ x := y \text{ op } z \]

1. Consult the address descriptor to determine \( L_y \) and \( L_z \)
2. Invoke `getreg()` to determine the location \( L_x \); update the address descriptor for \( x \); or update the register descriptor, if \( L_x \) is a register
3. If \( L_x = L_y \) or \( L_x = L_z \), generate “\( \text{op } L_y, L_z \)” or “\( \text{op } L_z, L_x \)”
4. Otherwise, generate “\( \text{mov } L_y \to L_x \)” then “\( \text{op } L_z \to L_x \)”;
   example: \( R_3 := R_5 + \text{Mem}[r_7] \) is mapped to: \( \text{mov } r_3<-r_5; \text{add } r_3<[r_7] \) // here destination on left
5. If the current values of \( y \) and/or \( z \) have no next uses, and are not live on block-exit, and are registers, mark in register descriptor as free

\[ x := \text{op } y \]
Similarly handled
A Simple CodeGen Algorithm, Cont’d

$\mathbf{a := b[i]}$

- $b$ in register $%Rb$
- $\text{ld} \ [\%rb+i], a$

$Lb$ is address/offset in: reg, mem, stack

- $b$ is memory addr $Mb$
- $\text{mov} \ Mb, \%r$
- $\text{mov} \ %fp+Sb, \%r$
- $\text{ld} \ [\%r+i], a$

If $x$ relop $y$ goto $z$

1. Move $x$ and $y$ to registers, if they are not there yet
2. Generate: $\text{comp<rel>} \ %rx, %ry \ // \ z, eq, ne, gt$
3. Generate: $\text{jumptrue} \ Lz \ //\text{assumes flag set}$

$\mathbf{a[i] := b}$

- $a$ in register $%Ra$
- $\text{st} \ b, [\%ra+i]$

$La$ is address/offset in: reg, mem, stack

- $a$ is mem addr $Ma$
- $\text{mov} \ Ma, \%r$
- $\text{mov} \ %fp+Sa, \%r$
- $\text{st} \ b, [\%r+i]$

- $a$ is stack offset $Sa$
- $\text{st} \ b, [\%r+i]$
Example: With Instruction Destinations RHS

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>Code Gen’d</th>
<th>Reg Descriptor</th>
<th>Address Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>t := a-b</td>
<td>mov a, %r0</td>
<td>%r0 contains t</td>
<td>t in %r0</td>
</tr>
<tr>
<td></td>
<td>sub b, %r0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>u := a-c</td>
<td>mov a, %r1</td>
<td>%r0 contains t</td>
<td>t in %r0</td>
</tr>
<tr>
<td></td>
<td>sub c, %r1</td>
<td>%r1 contains u</td>
<td>u in %r1</td>
</tr>
<tr>
<td>v := t+u</td>
<td>add %r1, %r0</td>
<td>%r0 contains v</td>
<td>v in %r0; t is dead</td>
</tr>
<tr>
<td></td>
<td>%r1 contains u</td>
<td></td>
<td>u in %r1</td>
</tr>
<tr>
<td>d := v+u</td>
<td>add %r1, %r0</td>
<td>%r0 contains d</td>
<td>d in %r0; v is dead</td>
</tr>
<tr>
<td></td>
<td>mov %r0, d</td>
<td>%r0 contains d</td>
<td>d is in memory</td>
</tr>
</tbody>
</table>

PSU CS322 HM 10
CodeGen with Dynamic Programming

Goal: Generate optimal code for broad class of register machines

*Machine Model:*

- $k$ interchangeable registers $r_0, r_1, \ldots, r_{k-1}$.
- Instructions are of the form $r_i := E$, where $E$ is an expression containing operators, registers, and memory locations (denoted $M$).
- Every instruction has an associated cost, measured by $C()$

Cost Vector: $C(E) = (c_0 \ c_1 \ \cdots \ c_r)$ — it’s defined for an expression $E$, where:

- $C_0$: cost of computing $E$ into memory, with the use of unbounded number of regs
- $C_i$: cost of computing $E$ into a register, with the use of up to $i$ regs
Dynamic Programming (DP)

Example: Consider multiplying a sequence of matrices,

\[ r := a \times b \times c \times d \times e \]
\[ [3 \times 6] [3 \times 5] [5 \times 8] [8 \times 1] [1 \times 2] [2 \times 6] \]

Assume the cost for multiplying an \([m \times n]\) matrix and an \([n \times l]\) matrix is \(O(m \times n \times l)\)

Question: What is the best order to multiply the sequence?

• A naive solution is to go from left to right:
  \[ r := ( ( (a \times b) \times c) \times d) \times e ) \text{ total cost } = 186 \]

• A better solution is:
  \[ r := (a \times (b \times c)) \times (d \times e) ) \text{ total cost } = 85 \]

DP Idea: 1. Try every possible combination 2. Avoid redundant Computations to contain compile-time
Dynamic Programming, Cont’d

• The cost of computing a sequence $s$ based on a particular partition $s = s_1s_2$ can be expressed as:
  $$c(s_1 | s_2) = c(s_1) + c(s_2) + c_{12}$$

• Create a table to record the lowest cost for computing each subsequence of $abcde$:
  $$c(ab), c(bc), c(cd), c(de), c(abc), c(bcd), c(cde), \ldots$$

• Divide and conquer — consider all four possible ways to partition the sequence $abcde$:
  $$c(a|bcde), c(ab|cde), c(abc|de), c(abcd|e)$$

• Recursively find the lowest cost for each case, then
  $$c(abcde) = \min(c(a|bcde), c(ab|cde), c(abc|de), c(abcd|e))$$
Applying DP toCodeGen

1. Compute bottom-up for each node $n$ of the expression tree $T$ an array $C$ of costs
2. Traverse $T$, using the cost vectors to determine which subtrees of $T$ must be computed into memory
3. Traverse $T$ and generate the final target code. The code for the subtrees computed into memory locations is generated first

Example: $(a - b) + c$  $(d/e)$

Target Machine Model:
Two registers: $r_0$, $r_1$
Uniform-cost instructions:

\[
\begin{align*}
    r_i &:= M \\
    r_i &:= r_i \text{ op } r_j \\
    r_i &:= r_i \text{ op } M \\
    r_i &:= r_j \\
    M &:= r_i
\end{align*}
\]

Cost Vector: $C = (c_0 \ c_1 \ c_2)$. 
CodeGen by Pattern Matching

Assume an IR tree language

• Defining tiles (i.e. patterns), each corresponds to an instruction

• Tiling the IR tree, covering the tree with non-overlapping tiles
## CodeGen by Pattern Matching, Cont’d

<table>
<thead>
<tr>
<th>Tree Patterns (Tiles)</th>
<th>Instructions</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TEMP )</td>
<td>-</td>
<td>ri</td>
</tr>
<tr>
<td>(BINOP + _ _)</td>
<td>add</td>
<td>ri &lt;- rj + rk</td>
</tr>
<tr>
<td>(BINOP - _ _)</td>
<td>sub</td>
<td>ri &lt;- rj - rk</td>
</tr>
<tr>
<td>(BINOP * _ _)</td>
<td>mul</td>
<td>ri &lt;- rj × rk</td>
</tr>
<tr>
<td>(BINOP / _ _)</td>
<td>div</td>
<td>ri &lt;- rj / rk</td>
</tr>
<tr>
<td>(BINOP + _ (CONST _))</td>
<td>addi</td>
<td>ri &lt;- rj + c</td>
</tr>
<tr>
<td>(BINOP + (CONST _) _)</td>
<td>subi</td>
<td>ri &lt;- rj - c</td>
</tr>
<tr>
<td>(MEM (BINOP + _ (CONST _)))</td>
<td>load</td>
<td>ri &lt;- M[rj + c]</td>
</tr>
<tr>
<td>(MEM (BINOP + (CONST _) _))</td>
<td>movem</td>
<td>M[rj ] &lt;- M[ri ]</td>
</tr>
<tr>
<td>(MEM _ (MEM _))</td>
<td>movem</td>
<td>M[rj ] &lt;- M[ri ]</td>
</tr>
<tr>
<td>(MOVE (MEM _ (MEM _))</td>
<td>store</td>
<td>M[rj + c] &lt;- ri</td>
</tr>
</tbody>
</table>
Sample: Pattern Matching

(MOVE (MEM (BINOP + (MEM (BINOP + (TEMP tFP) (CONST a)))
  (BINOP MUL (TEMP i) (CONST 4))))
  (MEM (BINOP + (TEMP tFP) (CONST x))))

Find Pattern, and then map to; destination Lhs except store:

load  r1 <- M[ fp + a ] // address of a[]
load  r2 <- i       // index i
mul   r2 <- r2 * 4  // i scaled: * 4, r2 sead
add   r1 <- r1 + r2 // address a[i] in r1
load  r2 <- M[ fp + x ] // formal param? in r2
store r2 -> M[ r1 ] // store param at M( a[i] )
                // r1 and r2 dead: re-use
Finding the Best Tiling

• We can associate a cost to each tile, then we can quantify the quality of a tiling of an IR tree
• Best tiling corresponds to instruction sequence of lowest cost
• Two Levels of Optimality:
  – Optimal Tiling: no two adjacent tiles can be combined into a single tile of lower cost. A greedy algorithm can be used to find an optimal tiling
  – Optimum Tiling: the tiles sum to the lowest possible value
  – Dynamic programming technique can be applied to tiling to find the optimum tiling for an IR tree
Maximal Munch Algorithm

For finding an optimal tiling:
1. Start at the tree’s root, find the “largest” fitting tile (which covers the root and maybe several other nodes near the root). Several subtrees may result from this.
2. Repeat same step for each subtree.
   “Largest” means most nodes. If two tiles of equal size match at the root, the choice between them is arbitrary.
Dynamic Programming Algorithm

Finding the optimum tiling: Algorithm assigns a cost to every tree node, which is the sum of the instructions costs of the best instruction sequence that can tile the subtree rooted at that node.

Works bottom-up. For each tile t of cost c that matches at node n, there will be zero or more subtrees si corresponding the leaves of the tile. Assume the cost for the subtree si is ci (which has already been computed), then the total cost of matching tile t at node n is c + \sum_{i} c_i.

1. Start at the leaves. For each node n, find all the tiles that matches at n, and compute their costs. The tile with the minimum cost is chosen, and the (minimum) cost is assigned to n.

2. Repeat this step for other nodes, until the root node’s cost is computed.

3. Now the algorithm goes top-down to emit instructions by calling Emission(root). Emission defined as follows: Emission(node n) — for each leaf li of the tile selected at node n, recursively perform Emission(li). Then emit the instruction matched.
Sample: Dynamic Programming

(MEM (BINOP + (CONST 1) (CONST 2)))

- A CONST node can be matched with only an addi instruction with cost 1. So the two CONST nodes each gets assigned a value of 1.
- At the BINOP + node, several matches are possible:

<table>
<thead>
<tr>
<th>Tile</th>
<th>Inst</th>
<th>T. Cost</th>
<th>L. Cost</th>
<th>T. Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BINOP +**)</td>
<td>add</td>
<td>1</td>
<td>1+1</td>
<td>3</td>
</tr>
<tr>
<td>(BINOP +*(CONST *))</td>
<td>addi</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(BINOP +(CONST*)*)</td>
<td>addi</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- The second tile is chosen and the minimum cost of 2 is assigned to the BINOP + node
Sample: Dynamic Programming, Cont’d

- At the MEM node, also several matches are possible:

<table>
<thead>
<tr>
<th>Tile</th>
<th>Inst</th>
<th>T. Cost</th>
<th>L. Cost</th>
<th>T. Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MEM *)</td>
<td>load</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(MEM (BINOP+*(CONST <em>))</em>)</td>
<td>load</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(MEM (BINOP+(CONST <em>)</em>))</td>
<td>load</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- Either of the last two matches will be optimum.
- The instructions emitted for this tree are:

```
addi r1 <- r0 + 1
load r1 <- M[r1 + 2]
```
Array Indexing

- Needs detailed symbol table information for each dim:
  - Low-bound, high-bound, element size, element type
- Element type is “array” for all but last dimension
- Element size for whole array is total size
- Subscript as l-value leaves the address, subscript as r-value dereferences memory once to get value
- Code generation for $a[ \text{inx } 0, \text{inx } 1, \ldots \text{inx } n-1 ]$:
  - Generate address of array $a[ ]$, named $addr$
  - For each index $\text{inx}_i$
    - Generate code for the expression $\text{inx}_i$
    - Bounds check against low- and high-bound - optional
    - Subtract low-bound
    - Multiply expression by element size
    - Add to $addr$
  - When done, and if r-value needed: gen code to dereference
Call Return

- Activation Record (AR) identified by frame pointer fp
- Other environments call this the base pointer: bp
- Stack marker (SM) holds: return value (unless in reg), static link (skip), dynamic link, return address (last, top)
- Actual parameters create initial values of formals; are pushed onto stack.
- Why do they have to be pushed in the reverse order from source?
- Formals on one side of SM, fp- (if growing stack increases addresses)
- Locals on other side of SM, fp+
- Saved regs on top of locals; number known per function = AR
- Temps on top of locals
- At call:
  - Reserve word for return value –unless returned in register
  - Call instruction: pushes return address; now SM holds 2 words
  - push fp –now sp points to “dynamic link”; now SM holds 3 words
  - fp = sp –the dynamic link is established
  - Allocate space for locals, regs, temp
  - sp has new value; AR includes space for formals, AR, locals, regs, temps