From IR to ASM: Challenges

- Register allocation
- Mapping IR to ASM operations
  - what instruction(s) should be used to implement an IR operation?
  - how do we translate code sequences?
- Memory allocation
- Call/return of routines
  - managing activation records
- Architecture-specific optimizations

Control Flow Graph

- Break the IR into basic blocks
- A basic block is a sequence of instructions with
  - single entry (to first instruction), no jumps to the middle of the block
  - single exit (last instruction)
  - code executes as a sequence from first instruction to last instruction without any jumps
- Edge from one basic block B1 to another block B2 when the last statement of B1 may jump to B2
Control Flow Graph: Example

Control Flow Graph: Definition

Creating Basic Blocks

Example

Simple Code Generation
Simple Code Generation

• Assume machine instructions of the form
  ▪ LD reg, mem
  ▪ ST mem, reg
  ▪ OP reg, reg, reg

• Assume that we have all registers available for our use
  ▪ Ignore registers allocated for stack management
  ▪ Treat all registers as general-purpose

Simple Code Generation

• Translate each 3AC instruction separately

• For each register, a register descriptor records the variable name(s)
  whose current value is in that register
  ▪ we use only those registers that are available for local use within a basic block
  ▪ as code generation progresses, each register will hold the value of zero or more variables

• For each program variable, an address descriptor records the location(s) where the current value of the variable can be found
  ▪ the location may be a register, a memory address, a stack offset, or some set of these
  ▪ information can be stored in the symbol-table entry for that variable

For each three-address statement \( x := y \) \( op \) \( z \),

1. Invoke \( \text{getreg}(x := y \) \( op \) \( z \) \) to select registers \( R_x, R_y, R_z \)
2. If \( R_y \) does not contain \( y \), issue: \( \text{LD} R_y, y' \) for a location \( y' \) of \( y \)
3. If \( R_z \) does not contain \( z \), issue: \( \text{LD} R_z, z' \) for a location \( z' \) of \( z \)
4. Issue the instruction \( \text{OP} R_x, R_y, R_z \)

The function \( \text{getreg} \) is not defined yet — for now think of it as an oracle that gives us a register per variable

Updating the Descriptors

1. For the instruction \( \text{LD} R_x, x \)
   ▪ change register descriptor for \( R_x \) so it holds only \( x \)
   ▪ change address descriptor for \( x \) by adding \( R_x \) as an additional location
2. For the instruction \( \text{ST} x, R \)
   ▪ change the address descriptor for \( x \) to include its own memory location
3. For an operation such \( \text{OP} R_x, R_y, R_z \) originating form a 3AC instruction \( x = y \) \( op \) \( z \)
   ▪ change register descriptor for \( R_x \) so that it holds only \( x \)
   ▪ change address descriptor for \( x \) so that its only location is \( R_x \), note that the memory location for \( x \) is not now in the address descriptor for \( x \)
   ▪ remove \( R_z \) from the address descriptor of any variable other than \( x \)
4. When we process a copy statement \( x = y \)
   ▪ generate load for \( y \) into register \( R_y \) and update descriptors accordingly (rule 1)
   ▪ add \( x \) to the register descriptor for \( R_y \)
   ▪ change the address descriptor for \( x \) so that its only location is \( R_y \)

Example

\[
\begin{align*}
R_1 & = A - B \\
& \text{LD} R_1, A \\
& \text{LD} R_2, B \\
& \text{SUB} R_1, R_2, R_1 \\
R_2 & = A - C \\
& \text{LD} R_2, A \\
& \text{LD} R_3, B \\
& \text{SUB} R_2, R_3, R_2 \\
R_3 & = t + u \\
& \text{ADD} R_3, R_1, R_3 \\
& \text{ADD} R_3, R_2, R_3 \\
& \text{ADD} R_3, R_2, R_3 \\
& \text{ADD} R_3, R_2, R_3 \\
\end{align*}
\]
Example

Variable Liveness

- A statement \( x = y + z \):
  - defines \( x \)
  - uses \( y \) and \( z \)
- A variable \( x \) is live at a program point if its value is used at a later point, prior to any (re-)definition of \( x \)

1: \( y = 42 \)
2: \( z = 73 \)
3: \( x = y + z \)
4: print\( x \)

\( x \) live (4), \( y \) dead, \( z \) dead
\( x \) dead, \( y \) live (3), \( z \) live
\( x \) dead, \( y \) live (3), \( z \) dead
\( x \) dead, \( y \) dead, \( z \) dead

Computing Liveness Information

- Between basic blocks – data-flow analysis (next lecture)
- Within a single basic block?
  - Idea
    - use symbol table to record next-use information
    - scan basic block backwards
    - update next-use for each variable

Computing Liveness Information

- INPUT: A basic block \( B \) of three-address statements. Symbol table initially shows all non-temporary variables in \( B \) as being live on exit.
- OUTPUT: For each statement \( \{ x = y + z \} \) in \( B \), liveness and next-use information of \( x, y, \) and \( z \) at \( (\text{just after}) \) \( l \).
- Start at the last statement in \( B \) and scan backwards
  - At each statement \( \{ x = y + z \} \) in \( B \), we do the following:
    1. Attach to \( l \) the information just found in the symbol table regarding the next use and liveness of \( x, y, \) and \( z \).
    2. In the symbol table, set \( x \) to “not live”.
    3. In the symbol table, set \( y \) and \( z \) to “live” and their next use to \( i \).

Computing Liveness Information
Computing Liveness Information

- Start at the last statement in B and scan backwards.
  - At each statement: $x = y + z$ in B, we do the following:
    1. Attach to i the information currently found in the symbol table regarding the next use and liveness of $x$, $y$, and $z$.
    2. In the symbol table, set $x$ to "not live".
    3. In the symbol table, set $y$ and $z$ to "live" and their next use to i.

can we change the order between steps 2 and 3 in the algorithm?

Local Register Allocation

- Simple rules
  - If $y$ is currently in a register, pick a register already containing $y$ as $R_y$. No need to load this register.
  - If $y$ is not in a register, but there is a register that is currently empty, pick one such register as $R_y$ and load it.
- Complicated case
  - $y$ is not in a register, but there is no free register.

Local Register Allocation

- Instruction: $x = y + z$
- Consider the case where:
  - $y$ is not in a register, no free register
  - $R$ is an occupied register holding value of a variable $v$
- Possibilities
  - if the value $v$ is available somewhere other than $R$, we can allocate $R$ to be $R_y$
  - if $v$ is $x$, the value computed by the instruction, we can use it as $R_y$ (it is going to be overwritten anyway)
  - if $v$ is not used later, we can use $R$ as $R_y$
  - otherwise: spill the value to memory by $ST\ v, R$

Global Register Allocation

- So far we assumed that register values are written back to memory at the end of every basic block.
- Want to save load/stores by keeping frequently accessed values in registers (e.g., loop counters).

Global Register Allocation

- Idea: compute "weight" for each variable.
  - each use of $v$ in $B$ prior to any definition of $v = 1$ point.
  - each variable $v$ computed in $B$ and live on exit = 2 points (as we save the store/load between blocks).

$$\text{cost}(v) = \sum_B \text{use}(v, B) + 2 \cdot \text{live}(v, B)$$

- after computing weights, allocate registers to the "heaviest" values.
Register Allocation by Graph Coloring

- Address register allocation by
  - Liveness analysis
  - Reduction to graph coloring

- **Main idea**
  - Register allocation = coloring of an interference graph
  - Every node is a variable
  - Edge between variables that “interfere” = are both live at the same time
  - Number of colors = number of registers
Register Allocation by Graph Coloring

- Variables that interfere with each other cannot be allocated the same register
- Graph coloring
  - classic problem: how to color the nodes of a graph with the lowest possible number of colors such that no two adjacent nodes have the same color
  - bad news: problem is NP-complete
  - good news: there are pretty good heuristic approaches

Heuristic Graph Coloring

- Idea: color nodes one by one, coloring the “easiest” node last
- “Easiest nodes” are ones that have lowest degree
  - fewer conflicts
- Algorithm at high-level
  - find a lowest-degree node
  - remove lowest-degree node from the graph
  - color the reduced graph recursively
  - re-attach the that node
Heuristic Graph Coloring

- Two sources of non-determinism in the algorithm
  - choosing which of the (possibly many) nodes of lowest degree should be detached
  - choosing a free color from the available colors
Assembly Code

Now for some Assembly Code.

Intel IA-32 Assembly

- Going from Assembly to Binary...
  - Assembling
  - Linking

- AT&T syntax vs. Intel syntax
- We will use AT&T syntax
  - matches GNU assembler (GAS)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>AT&amp;T</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter order</td>
<td>Source, destination</td>
<td>Destination, source</td>
</tr>
<tr>
<td>Parameter width</td>
<td>Parameter is suffixed with a letter indicating the size of the operands (e.g., &quot;b&quot; for byte, &quot;w&quot; for word, &quot;d&quot; for double word, &quot;q&quot; for quad-word)</td>
<td>Parameter is derived from the name of the register that is used</td>
</tr>
<tr>
<td>Symbols</td>
<td>Immediate values prefixed by &quot;$&quot;, registers prefixed by &quot;%&quot;</td>
<td>The assembler automatically detects the type of symbols; i.e., if they are registers, immediates, or constants.</td>
</tr>
<tr>
<td>Effective addresses</td>
<td>Uses special syntax: DISP(BASE,INDEX,SCALE)</td>
<td>Uses variables, and need to be in square brackets; additionally, size keywords like <code>byte</code>, <code>word</code>, <code>dword</code> have to be used.</td>
</tr>
</tbody>
</table>

```
movl mem_location(%ebx,%ecx,4), %eax
mov eax, dword [ebx + ecx*4 + mem_location]
```

IA-32 Registers

- Eight 32-bit general-purpose registers
  - EAX – accumulator for operands and result data. Used to return value from function calls.
  - EBX – pointer to data. Often used as array-base address
  - ECX – counter for string and loop operations
  - EDX – I/O pointer (GP for us)
  - ESI – GP and source pointer for string operations
  - EDI – GP and destination pointer for string operations
  - EBP – stack frame (base) pointer
  - ESP – stack pointer

- EFLAGS register
- EIP (instruction pointer) register
- Six 16-bit segment registers
- ... (ignore the rest for our purposes)

Not all registers are born equal

- EAX – Required operand of MUL, MSU, DIV, and IDIV instructions
- EDX – Contains the result of these operations
- ESI, EDI
  - ESI – Required source pointer for string instructions
  - EDI – Required destination pointer for string instructions
- Destination Registers of Arithmetic operations
  - EAX, EBX, ECX, EDX
- EBP – stack frame (base) pointer
- ESP – stack pointer
IA-32 Addressing Modes

- Machine-instructions take zero or more operands
- Source operand
  - Immediate
  - Register
  - Memory location
- Destination operand
  - Register
  - Memory location

Immediate and Register Operands

- Immediate
  - Value specified in the instruction itself
  - GAS syntax — immediate values preceded by $
    - add $4, %esp
- Register
  - Register name is used
  - GAS syntax — register names preceded with %
    - mov %esp, %ebp

Memory and Base Displacement Operands

- Memory operands
  - Value at given address
  - GAS syntax — parentheses
    - mov (%eax), %eax
- Base displacement
  - Value at computed address
  - Address computed out of
    - base register, offset register, scale factor, displacement
    - address = base + (offset*scale) + disp
    - movl $42, $2(%eax)
    - movl $42, $1(%eax,%ecx,4)

Base Displacement Addressing

- Array Base Reference
  - base = %ecx
  - offset = %ebx
  - scale = 4
  - address = base + (index*scale) + displacement
  - movl (%ecx,%ebx,4), %eax
    - base = %ecx
    - offset = %ebx
    - scale = 4
    - address = %ecx + (3*4) + 0 = %ecx + 12

Instruction Selection

- How do we choose which instructions to emit from our IR?
- Greatly depends on how we design our IR.
  - We see two examples in this lecture:
    - Tree-structured IR — Tiling
    - 3AC sequence — Peephole Matching
Canonical IR — Tiling

Example:

\[ x = x + 1; \]

\[
\text{MEM}\quad \text{MEM}\quad \text{MEM}
\]

\[
\text{FP}\quad \text{XP}\quad \text{YP}
\]

\[
\text{MOVE}\quad \text{MOVE}\quad \text{MOVE}
\]

\[
\text{MEM}\quad \text{MEM}\quad \text{MEM}
\]

\[
\text{FP}\quad \text{FP}\quad \text{FP}
\]

\[
\text{MOVE}\quad \text{MOVE}\quad \text{MOVE}
\]

Tiles

- Capture compiler’s understanding of the instruction set
- Tile = sequence of instructions that update a fresh temporary (may need extra mov’s) and associated IR tree
- Each outgoing edge represents a temporary

CISC vs. RISC Tiling

Example Tiles for “+”

Based on Intel Architecture Manual, Vol 2, 3-17

Computing a Tiling

- Maximal Munch — a greedy approach
  - Start at statement root
  - Find largest tile covering node, matching all children and outgoing edges
  - Invoke recursively on temporaries dangling from tile
  - Generate code for tile
    - (code for children will have been generated in recursive calls)
Timing Model

- **Idea**: associate cost with each tile (proportional to # cycles to execute)
  - sum of costs approximates execution time

```
  MOVE
  MEM
  FP

  Total cost = 5
```

Computing Optimum Tiling

- **Goal**: find minimum total cost tiling of tree
- **Algorithm**: for every node, find minimum total cost tiling of that node and sub-tree.
- **Lemma**: once minimum cost tiling for all descendants of a node is known, one can find minimum cost tiling of the node (sub-tree) by trying out all possible tiles matching the top
  ⇒ start from leaves, work upward to top node

Dynamic Programming

Example:

```
  movl a(%ebp), t1  
movl i(%ebp), t2  
movl (t1,t2,4), t3
```

Peephole Matching

- **Basic idea**: discover local improvements locally
  - Look at a small set of adjacent operations
  - Move a small sliding window (“peephole”) over code and search for improvement
- **Classic examples**:
  - store followed by load
  - algebraic identities
  - jump to jump

Peephole Matching

- **How to implement it?**

  Modern instruction selectors break problem into three tasks: 
  (Davidson, 1989)
Peephole Matching

Expander

- Turns IR code into a low-level IR (LLIR) such as RTL
- Operation-by-operation, template-driven rewriting
- LLIR form includes all direct effects
- Significant, albeit constant, expansion of size

Simplifier

- Looks at LLIR through window and rewrites is
- Uses forward substitution, algebraic simplification, local constant propagation, and dead-effect elimination
- Performs local optimization within window

Matcher

- Compares simplified LLIR against a library of patterns
- Picks low-cost pattern that captures effects
- Must preserve LLIR effects, may add new ones (e.g., set cc)
- Generates the assembly code output

Peephole Matching — Example

Original IR Code

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>-</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
</tr>
</tbody>
</table>

LLIR Code

\[
\begin{align*}
\text{f}_2 &= 2 \\
\text{f}_3 &= y \\
\text{f}_4 &= \text{op} \times \text{f}_1 \\
\text{f}_5 &= \text{op} \times \text{f}_1 \\
\text{f}_6 &= \text{op} \times \text{f}_1 \\
\text{f}_7 &= \text{op} \times \text{f}_1 \\
\text{f}_8 &= \text{op} \times \text{f}_1 \\
\text{f}_9 &= \text{op} \times \text{f}_1 \\
\text{f}_{10} &= \text{op} \times \text{f}_1 \\
\text{f}_{11} &= \text{op} \times \text{f}_1 \\
\text{f}_{12} &= \text{op} \times \text{f}_1 \\
\text{f}_{13} &= \text{op} \times \text{f}_1 \\
\text{f}_{14} &= \text{op} \times \text{f}_1 \\
\text{f}_{15} &= \text{op} \times \text{f}_1 \\
\text{f}_{16} &= \text{op} \times \text{f}_1 \\
\text{f}_{17} &= \text{op} \times \text{f}_1 \\
\text{f}_{18} &= \text{op} \times \text{f}_1 \\
\text{f}_{19} &= \text{op} \times \text{f}_1 \\
\text{f}_{20} &= \text{op} \times \text{f}_1 \\
\end{align*}
\]

LLIR Code

\[
\begin{align*}
\text{f}_2 &= 2 \\
\text{f}_3 &= y \\
\text{f}_4 &= \text{f}_1 \times \text{f}_1 \\
\text{f}_5 &= \text{f}_1 \times \text{f}_1 \\
\text{f}_6 &= \text{f}_1 \times \text{f}_1 \\
\text{f}_7 &= \text{f}_1 \times \text{f}_1 \\
\text{f}_8 &= \text{f}_1 \times \text{f}_1 \\
\text{f}_9 &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{10} &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{11} &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{12} &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{13} &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{14} &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{15} &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{16} &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{17} &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{18} &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{19} &= \text{f}_1 \times \text{f}_1 \\
\text{f}_{20} &= \text{f}_1 \times \text{f}_1 \\
\end{align*}
\]
Peephole Matching — Example

LLIR Code
\[
\begin{align*}
&f_{13} := (b_p + @y) \\
&f_{14} := 2 \cdot f_{13} \\
&f_{15} := f_{15} - f_{14} \\
&*(b_p + @w) := f_{13}
\end{align*}
\]

Assembly [LL/LOC] Code
\[
\begin{align*}
&\text{load} f_{13} \text{ \{\text{\#y} \}} \\
&\text{mul} f_{13}, f_{13} := w_{14} \\
&\text{sub} f_{13}, f_{14} := w_{15} \\
&\text{store} f_{13} \text{ \{\text{\#w} \}}
\end{align*}
\]

Making Peephole Matching Work

Details:
- LLIR is largely machine independent [RTL]
- Target machine described as LLIR → ASM pattern
- Actual pattern matching
  - Use a hand-coded pattern matcher [gcc]
  - Turn patterns into grammar & use LR parser (VPO)
- Several important compilers use this technology
- It seems to produce good portable instruction selectors

Key strength appears to be late low-level optimization

Summary

- Register allocation
  - "Local" = with a basic block
  - "Global" = across blocks (in a single function)
- Instruction selection
  - Use Tiling to match portions of IR tree
    - Greedy approach is fast
    - Dynamic programming more costly, but can find optimal tiling w.r.t. timing model
  - Use Peephole matching directly on 3AC