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

Simple Code Generation

- Registers
  - used as operands of instructions
  - can be used to store temporary results
  - can (should) be used as loop indexes due to frequent reads and increments
  - used to manage administrative info (e.g., runtime stack)
- Number of registers is limited
  ⇒ Need to allocate them in a clever way
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; we assume that initially, all register descriptors are empty
  — 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

Simple Code Generation

For each three-address statement \( x := y \ o p \ z \),
1. Invoke getreg \((x := y \ o p \ z)\) to select registers \( R_x, R_y, \) and \( R_z \)
2. If \( R_y \) does not contain \( y \), issue: \( LD \ R_y, y' \) for a location \( y' \) of \( y \)
3. If \( R_z \) does not contain \( z \), issue: \( LD \ R_z, z' \) for a location \( z' \) of \( z \)
4. Issue the instruction \( OP \ R_x, R_y, R_z \)

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 \( LD \ R, x \)
   a) Change register descriptor for \( R \) so it holds only \( x \)
   b) Change address descriptor for \( x \) by adding \( R \) as an additional location
2. For the instruction \( ST \ x, R \)
   a) Change the address descriptor for \( x \) to include its own memory location
3. For an operation such \( OP \ R_x, R_y, R_z \) originating form a 3AC instruction \( x = y \ o p \ z \)
   a) Change register descriptor for \( R_x \) so that it holds only \( x \)
   b) Change address descriptor for \( x \) so that its only location is \( R_x \); note that the memory location for \( x \) is not one of the address descriptors for \( x \)
   c) Remove \( R_x \) from the address descriptor of any variable other than \( x \)
4. When we process a copy statement \( x := y \)
   a) Generate load for \( y \) into register \( R_y \) and update descriptors accordingly (rule 1)
   b) Add \( x \) to the register descriptor for \( R_y \)
   c) Change the address descriptor for \( x \) so that its only location is \( R_y \)

Example

\[
\begin{array}{ccccccccc}
R1 & R2 & R3 & A & B & C & D & t & w \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{array}{ccccccccc}
R1 & R2 & R3 & A & B & C & D & t & w \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{array}{ccccccccc}
R1 & R2 & R3 & A & B & C & D & t & w \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{array}{ccccccccc}
R1 & R2 & R3 & A & B & C & D & t & w \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

A, B, C, D = variables in function scope
\( A := u \) = temporary in block
Example

R1 R2 R3 A B C D E F G H
\[\begin{array}{l}
A = D
u & u & u & u
R1 R2 R3 A B C D E F G H
\end{array}\]

R2 R3 A B C D E F G H

Example

t = A – B
u = A – C
v = t + u

A = D
D = v + u

Variable Liveness

- A statement \( x = y + z \):
  - \( x \) defines \( x \)
  - \( y \) and \( z \) 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 \( x \).

Computing Liveness Information

- 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

- \(\text{INPUT}\): A basic block \( B \) of three-address statements.
  - Symbol table initially shows all non-temporary variables in \( B \) as being live on exit.
- \(\text{OUTPUT}\): For each statement \( i: x = y + z \) in \( B \), liveness and next-use information of \( x \), \( y \), and \( z \) at (just after) \( i \).
- Start at the last statement in \( B \) and scan backwards
  - At each statement \( i: x = y + z \) in \( B \), we do the following:
    1. Attach to \( i \) the information previously 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

- Start at the last statement in \( B \) and scan backwards
  - At each statement \( i: x = y + z \) in \( B \), we do the following:
    1. Attach to \( i \) the information previously 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

- Start at the last statement in B and scan backwards
  - At each statement \( i : x = y + z \) 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 \( i \).

\[
\begin{align*}
x & = 1 \\
y & = x + 3 \\
z & = x' \times 3 \\
x & = x' \times z
\end{align*}
\]

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 \( R \) (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)

- 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
Example

\[ a = b + c \]
\[ d = d - b \]
\[ e = a + f \]

bcdf

\[ f = a - d \]

acde

cdef

\[ b = d + f \]

acdf

bcdef

\[ b = d + c \]

bcdef

b,c,d,e,f live

B1

B2

B3

B4

acdef

b,d,e,f live

1 use of a prior to def

f = a - d

acdf

bcdef

b,c,d,e,f live

B1

B2

B3

B4

acdef

b,d,e,f live

1 use of a prior to def

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

Example

 interference graph

Example

Example
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

Result:
3 registers for 6 variables
Can we do with 2 registers?

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

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)

AT&T vs. Intel Syntax

<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></td>
<td></td>
</tr>
</tbody>
</table>
  - 8 bits are suffixed with a
    letter indicating the size of the
    operands (e.g., “b” for byte, “w”
    for word, “l” for double word, “q”
    for quad word) |
    - Derived from the name of the
      register that is used |
| Symbols | Immediate values prefixed by “$”,
          registers prefixed by “%” | The assembler automatically detects
the type of symbols, i.e., if they are
registers, immediates, or constants. |
| Effective addresses | Special syntax: DISP(BASE,INDEX,SCALE) | The assembler automatically detects
the type of symbols, i.e., if they are
registers, immediates, or constants. |
  
  - Use +, *, and enclose in square brackets; additionally, size keywords like byte, word, or dword have to be used. |

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,MSD,LDIV and IDIV instructions
  - Contains the result of these operations
• EDX
  - Stores remainder of a DIV or IDIV instruction
  - EAX stores quotient
• 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
%ecx

base = %ecx
offset = %ebx
scale = 4

mov (%ecx,%ebx,4), %eax
address = base + (index*scale) + displacement
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:
    - Peephole Matching (suitable for 3AC)
    - Tiling (suitable for tree-structured IR)
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:**
  1. `*p := R1 R1 := *p`
  2. `R7 := R2 + 0` goto L10
  3. `R10 := R8 * R7 Lc` goto L10

  *store followed by load*  
  *algebraic identities*  
  *jump to jump*

---

**How to implement it?**

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

- **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 (e.g., setting flags)
  - Significant, albeit constant, expansion of size

- **Simplifier**
  - Looks at LLIR through window and rewrites it
  - 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
  - Generates the assembly code output
Peephole Matching — Example

**Original IR Code**

<table>
<thead>
<tr>
<th>op</th>
<th>F0</th>
<th>F1</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>y</td>
<td>1</td>
</tr>
</tbody>
</table>

$r_0 = t_{in}$

**LIR Code**

$r_0 = 2$
$r_1 = @y$
$r_2 = b * r_1$
$r_3 = *(r_2)$
$r_4 = r_0 * r_3$
$r_5 = &y$
$r_6 = b * r_5$

**Expand**

$r_7 = &y$
$r_8 = b * r_7$

**LLIR Code**

$r_0 := @y$
$r_1 := b * r_0$
$r_2 := *(r_1)$
$r_3 := r_0 * r_2$

**Simplify**

$r_4 := @y$
$r_5 := b * r_4$

**Assembly (x86) Code**

```assembly
movl @y(%ebp), %esi
movl %esi, %edi
shll $1, %edi
movl @x(%ebp), %ecx
movl %ecx, %edx
subl %edi, %edx
movl %edx, @w(%ebp)
```

**Steps of the Simplifier**

**Match**

$r_{13} := *(b + @y)$

$r_{14} := 2 * r_{13}$

$r_{17} := *(b + @x)$

$r_{18} := r_{17} – r_{14}$

$r_{19} := @w$

$r_{20} := b + r_{19}$

$r_{21} := *(b + @w)$

$r_{22} := r_{18}$
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 (VPQ)
- Several important compilers use this technology
- It seems to produce good portable instruction selectors

Key strength appears to be late low-level optimization

Canonical IR — Tiling

Example:
```plaintext
x = x + 1;
```

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

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`

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
Coming Up

Static Analysis