Local Register Allocation (and Assignment)

**Readings:** EaC 13.1-13.2, Appendix A (ILOC)

Local: within single basic block
Global: across procedure/function
Register Allocation

Part of the compiler’s back end

- Produce correct code that uses \( k \) (or fewer) registers
- Minimize added loads and stores
- Minimize space used to hold spilled values
- Operate efficiently
  \( O(n), O(n \log_2 n), \) maybe \( O(n^2) \), but not \( O(2^n) \)
• **register-register model**  
  → Values that *may safely reside* in registers are assigned to a unique virtual register (*alias analysis*)
  → Register allocation/assignment maps virtual registers to limited set of physical registers
  → Register allocation/assignment pass needed to make code “work”

• **memory-memory model**  
  → All values reside in memory, and are only kept in registers as briefly as possible (load operands from memory, perform computation, store result into memory)
  → Register allocation/assignment has to try to identify cases where values can be safely kept in registers
  → Safety verification is hard at the low levels of program abstraction
  → Even without register allocation/assignment, code will “work”
• register-register model
  → Values that may safely reside in registers are assigned to a unique virtual register (alias analysis; unambiguous values); there are different "flavors"

• memory-memory model
  → All program-named values reside in memory, and are only kept in registers as briefly as possible (load operands from memory, perform computation, store result back into memory)

```
a := 1
b := 2
c := a + b + 3
```

assumption: no aliasing
Consider a fragment of assembly code (or ILOC)

```
loadI 2 ⇒ r1  // r1 ← 2
loadAI r0, @y ⇒ r2  // r2 ← y
mult  r1, r2 ⇒ r3  // r3 ← 2 · y
loadAI r0, @x ⇒ r4  // r4 ← x
sub   r4, r3 ⇒ r5  // r5 ← x - (2 · y)
```

The Problem

- At each instruction, decide which values to keep in registers
  - Note: a value is a pseudo-register (virtual register)
  - Simple if \( |values| \leq |registers| \)
- Harder if \( |values| > |registers| \)
- The compiler must automate this process
Consider a fragment of assembly code (or ILOC)

address immediate

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Immediate</th>
<th>=&gt;</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI</td>
<td>2</td>
<td></td>
<td>=&gt;</td>
<td>r1</td>
<td>// r1 ← 2</td>
</tr>
<tr>
<td>loadAI</td>
<td>r0, 8</td>
<td></td>
<td>=&gt;</td>
<td>r2</td>
<td>// r2 ← y</td>
</tr>
<tr>
<td>mult</td>
<td>r1, r2</td>
<td></td>
<td>=&gt;</td>
<td>r3</td>
<td>// r3 ← 2 · y</td>
</tr>
<tr>
<td>loadAI</td>
<td>r0, 4</td>
<td></td>
<td>=&gt;</td>
<td>r4</td>
<td>// r4 ← x</td>
</tr>
<tr>
<td>sub</td>
<td>r4, r3</td>
<td></td>
<td>=&gt;</td>
<td>r5</td>
<td>// r5 ← x - (2 · y)</td>
</tr>
</tbody>
</table>

The Problem

At each instruction, decide which values to keep in registers

→ Note: a value is a pseudo-register (virtual register)

→ Simple if \(|values| \leq |registers|\)

→ Harder if \(|values| > |registers|\)

• The compiler must automate this process
The Task

• At each point in the code, pick the values to keep in registers
• Insert code to move values between registers & memory
  → No reordering transformations (leave that to scheduling)
• Minimize inserted code — both dynamic & static measures
• Make good use of any extra registers

Allocation versus assignment

• Allocation is deciding which values to keep in registers
• Assignment is choosing specific registers for values
• This distinction is often lost in the literature

The compiler must perform both allocation & assignment
Local Register Allocation

- **What’s “local”?** (as opposed to “global”)
  - A local transformation operates on basic blocks
  - Many optimizations are done locally

- **Does local allocation solve the problem?**
  - It produces decent register use inside a block
  - Inefficiencies can arise at boundaries between blocks

- **How many passes can the allocator make?**
  - This is a compile-time (“off-line”) problem (not done during program execution); typically, as many passes as it takes

- **memory-to-memory vs. register-to-register model**
  - Code shape and safety issues
Can we do this optimally? (on real code?)

**Local Allocation**
- Simplified cases $\Rightarrow \mathcal{O}(n)$
- Real cases $\Rightarrow \text{NP-Complete}$

**Global Allocation**
- NP-Complete for 1 register
- NP-Complete for $k$ registers
  (most sub-problems are NPC, too)

**Local Assignment**
- Single size, no spilling $\Rightarrow \mathcal{O}(n)$
- Two sizes $\Rightarrow \text{NP-Complete}$

**Global Assignment**
- NP-Complete

*Real compilers face real problems*
Basic Approach of Allocators

Allocator may need to reserve physical registers to ensure feasibility

- Must be able to compute memory addresses
- Requires some minimal set of registers, \( F \)
  \( \Rightarrow \) \( F \) depends on target architecture
- \( F \) contains registers to make spilling work
  \( \Rightarrow \) set \( F \) registers “aside” for address computation & instruction execution, i.e. these are not available for register assignment
- Note: \( F \) physical registers need to be able to support the pathological case where all virtual registers are spilled

What if \( k - |F| < |values| < k \)?
- The allocator can either
  \( \Rightarrow \) Check for this situation
  \( \Rightarrow \) Accept the fact that the technique is an approximation

Notation:
\( k \) is the number of registers on the target machine
Top-down allocator
- May use notion of “live ranges” of virtual registers
- Work from “external” notion of what is important
- Assign registers in priority order
- Register assignment remains fixed for entire basic block
- Save some registers for the values relegated to memory (feasible set F)

Bottom-up allocator
- Work from detailed knowledge about problem instance
- Incorporate knowledge of partial solution at each step
- Register assignment may change across basic block
- Save some registers for the values relegated to memory (feasible set F)
Live Ranges (live on exit)

Assume i and j are two instructions in a basic block

A value (virtual register) is live between its definition and its uses
• Find definitions ($x \leftarrow \ldots$) and uses ($y \leftarrow \ldots x \ldots$)
• From definition to last use is its live range
  → How many (static) definitions can you have for a virtual register?
• Can represent live range as an interval $[i,j]$ (in block)
  → live on exit

Let $\text{MAXLIVE}$ be the maximum, over each instruction $i$ in the block, of the number of values (virtual registers) live at $i$.
• If $\text{MAXLIVE} \leq k$, allocation should be easy
  → no need to reserve set of $F$ registers for spilling
• If $\text{MAXLIVE} > k$, some values must be spilled to memory

Finding live ranges is harder in the global case
More Register Allocation EaC 13.1 - 13.3
(Top-down and Bottom-Up Allocation)