

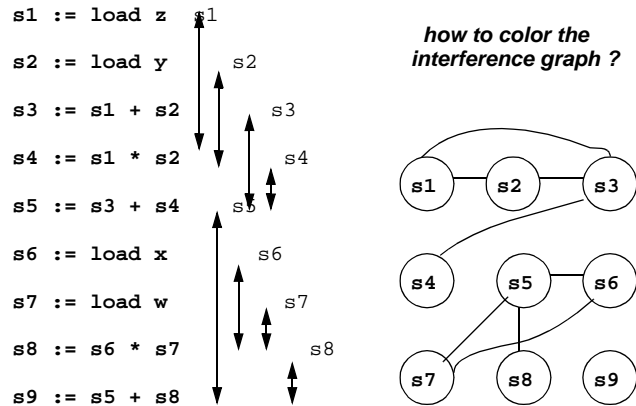
More on Machine-Code Generation

- **Problem:** given a target machine specification, how to translate the intermediate representations into **efficient** machine code ?
- **Solution** --- must take consideration of the machine architecture
 1. **Code Selection**
(emitting the machine code via **maximal-munch** or **dynamic programming**)
 2. **Register Allocation**
(global register allocation, spilling)
 3. **Instruction Scheduling**
(instruction scheduling, branch prediction, memory hierarchy optimizations)
- **Language Trends :** assembly -> C -> ... -> higher-level languages ?
- **Architecture Trends :** CISC -> RISC -> ... -> superscalar -> ?
- **Trends:** the bridging gap is the main **challenge** to compiler writers

Register Allocation

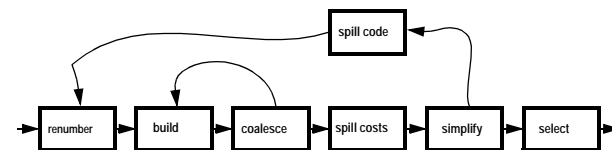
- **Register allocation** often works on the intermediate representations that are very much like the machine code.
- **Input:** intermediate code that references unlimited number of registers; **output:** rewrite the intermediate code so that it uses the limited registers available on the target machine --- the machine registers.
- **Standard Algorithm:** **Graph Coloring Register Allocation**
 Main idea: build a interference graph based on the live ranges of each identifiers; then color the interference graph.
 Example: **Yorktown Allocator** (by Chaitin et al. at IBM T.J.Watson)'
Briggs's Extension (by Briggs et al. at Rice Univ.)

Example: Register Allocation



Yorktown Allocator

- **Renumber:** name all identifiers uniquely, find out their live ranges.
- **Build:** construct the interference graph G.
- **Coalesce:** eliminating copying instructions, e.g., $x = y$
- **Spill Costs:** calculate the spill costs
- **Simplify:** (together with **Select**) color the graph (it is NP-complete!).
- **Select:** choose the actual colors (i.e., registers)
- **Spill Code:** insert the spill code



Yorktown Allocator (cont'd)

- **Build:** the interference graph characterizes the **interference** relation of live ranges: two live ranges **interfere** if there exists some point in the procedure and a possible execution of the procedure such that

1. both live ranges have been defined
2. both live ranges will be used, and
3. the live ranges have different values

- **Simplify and Select:** assuming there are k physical registers

In **Simplify**, the allocator repeatedly removes nodes with outer degree $< k$ from the graph and pushes them onto a stack.

In **Select**, the nodes are popped from the stack and added back to the graph --- a color is chosen for each node.

If **Simplify** encounters a graph containing only nodes of degree $\geq k$, then a node is chosen for spilling.

Yorktown Allocator (cont'd)

- **Choosing Spill Nodes:** based on the weight m_n for each node n

Chaitin's heuristics: $m_n = \text{cost}_n / \text{degree}_n$

Alternatives: $m_n = \text{cost}_n / (\text{degree}_n * \text{area}_n)$

where area_n is a function that quantifies the impact n has on other live ranges in the program, e.g., if it is used in a loop often, area_n is larger.

- **Spilling:** if v is spilled, a **store** is inserted after every definition of v , and a **load** is inserted before every use of v .

- **Bernstein et al.** later found no single **spilling-cost heuristics** completely dominates the other. They propose "**best of 3**" technique:

Just run the algorithm using three heuristics, then choose one with the best outcome.

Briggs's Extension

- **Simplify** removes nodes with degree $< k$ in an arbitrary order. If all remaining nodes have degree $\geq k$, a spill candidate is choosed and optimistically pushed on the stack also, hoping a color will be found later.
- **Select** may discover that it has no color for some node. In that case, it leaves the node uncolored and continues with the next node.
- If any nodes are uncolored, the allocator inserts spill code accordingly and rebuild the interference graph, and tries again.

