CSc 553
Principles of Compilation

06. Register Allocation

Saumya Debray

The University of Arizona

Tucson, AZ 85721

Register allocation

- Goals:
  - place frequently accessed values in registers.
  - reduce (minimize?) memory accesses.
- Interaction with code generation:
  - Code generation assumes an infinite no. of “virtual registers”.
  - *Register allocation*: determine which virtual registers get mapped to physical registers.
  - *Register assignment*: determine the actual mapping from physical registers to virtual registers.

Register allocation

Scope of register allocation:

Local (basic block level)

- simpler to implement
- limited performance benefits

Global (function level)

- implementation is more complex
- better performance benefits

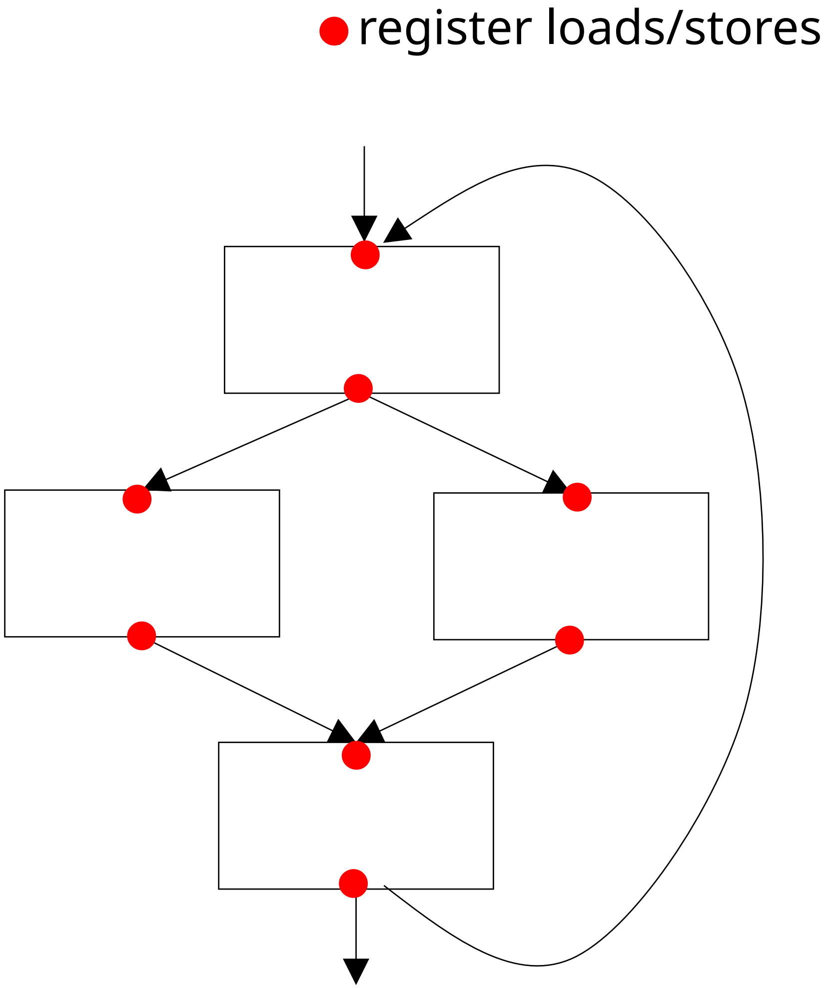
Local register allocation

Bottom-up local register allocation

- Consider the instructions in the basic block in order. When a register is needed for an operand:
  - if a free register is available, use it
  - otherwise, free up a [least-cost] register whose next use is furthest in the future:
    - generate code to store its contents into memory
    - update bookkeeping info accordingly
- At the end of the block, store (live) variables back into memory
- Bookkeeping info: register contents; location of variables

Issues with local register allocation

- Register loads and stores at every basic block boundary
- 80-20 rule suggests that this will typically result in a lot of memory traffic
  - performance gains from register usage are diluted



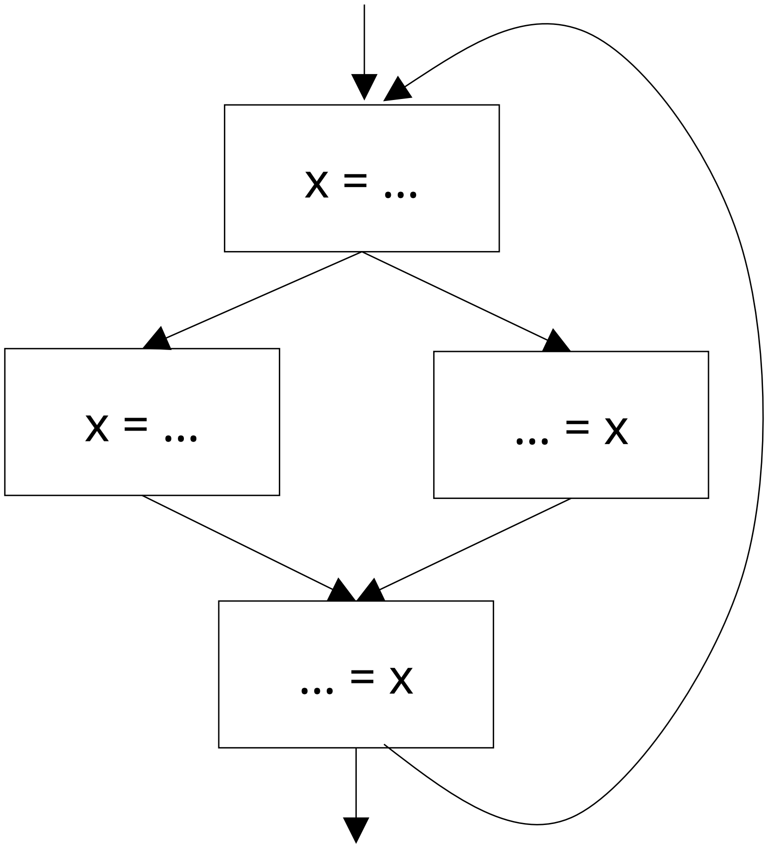
Global register allocation

Global register allocation

Main issues:

- definitions and uses of a variable should refer to the same register
  - *live ranges*

- different variables should not get mapped to the same register
  - *graph coloring*



Live ranges

Live ranges

A live range consists of a set of definitions D and uses U of a variable x, such that:

- for each use $u \in U$, every definition that can reach u is in D
- for each definition $d \in D$, every use that d can reach is in U

- Live ranges form the unit of global register allocation
  - each live range for a variable is mapped to a particular storage location (a specific register; or else memory)

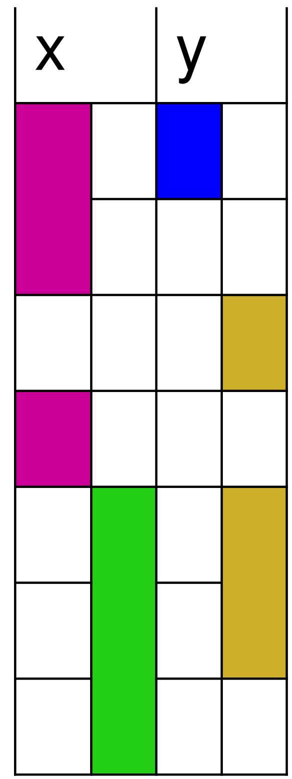
Live ranges

```
int f(...) {
    for (x = ...; x < n; x++) {
        ... use(x) ...
        ... use(x) ...
    }
    ... use(y) ...
    ... use(y) ...
    x = ...
    ... use(x) ...
    ... use(x) ...
```

Live ranges: Example

x = y + 1
z = x + a
y = u + w
v = x + 4
x = y * z
v = x - y
u = v / x

Live Ranges



Identifying Live Ranges

- Carry out *reaching definitions analysis.*
- For each definition d_x of a variable x, let $U(d_x)$ be the set of uses associated with d_x:

  $U(d_x)$ = { i | instruction i uses x and d_x reaches i }.

  $LR(d_x) = \{d_x\} \cup U(d_x)$. /* Live range = the definition + all its uses */
- repeat
  - if there are two definitions d_1 and d_2 of a variable x such that LR(d_1) and LR(d_2) overlap, merge LR(d_1) and LR(d_2).
    - i.e., set LR(d_1) and LR(d_2) to $LR(d_1) \cup LR(d_2)$.

  until no further merging occurs.

Register interference graph

Register allocation via graph coloring

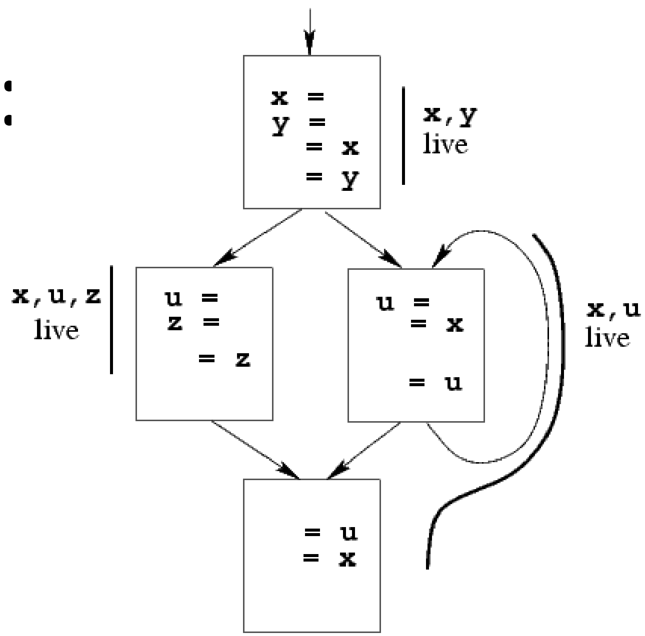
- Graph coloring is a systematic way of allocating registers and managing spills.
- Consists of two passes:
  1. Target machine instructions are selected as though there is an unbounded no. of symbolic registers.
  2. Physical registers are assigned to symbolic registers in a way that minimizes spill costs.

This is done by constructing a ***register interference graph*** for the function being compiled, and then k-coloring this graph (k = no. of available registers).

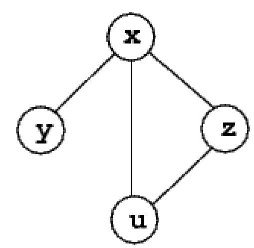
Register interference graphs

- Nodes ≈ live ranges
- There is an edge between two nodes if they can be simultaneously live [“interference”]

Example:



Control flow graph



Register interference graph

Graph coloring: Overview

- <u>*The Graph Coloring problem*</u>:

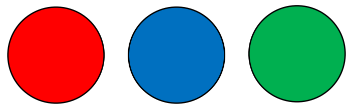
  Given a graph G = (V, E) and a fixed finite set of colors C, find a mapping *color*: V → C satisfying:

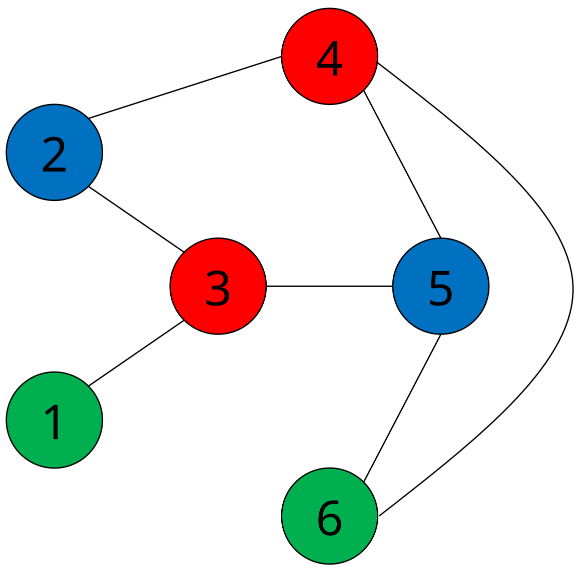
  for every $(a, b) \in E$: $color(a) \neq color(b)$

- For register allocation:
  - the set of colors corresponds to the set of registers
  - If there are k registers, the no. of colors = k.

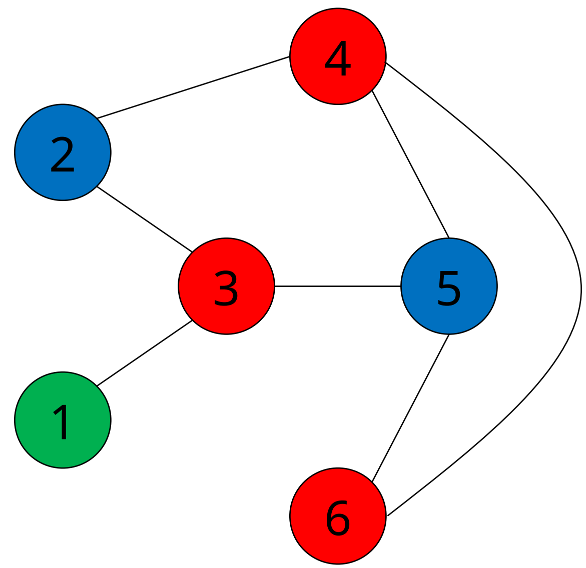
- NP-complete in general
  - no known algorithm that will always compute the optimal solution efficiently
  - resort to heuristics that work well in practice

Graph coloring: Example

Colors = 



Coloring 1



Coloring 2

Graph coloring: Heuristic

A coloring heuristic often used for register allocation:

repeat

delete each node with fewer than *k* neighbors

/ we can always find a color for these nodes later */*

until either:

1. the resulting graph is empty:

   work backwards to produce a *k*-coloring of the original graph; or

2. every node has $\geq k$ neighbors:

   pick a node *x* to spill, delete *x* from the graph, and repeat the above process.

Graph coloring register allocation: Issues

- Identifying live ranges
- Constructing the interference graph
- Choosing spill nodes.
  - need to estimate spill costs.

Constructing the Interference Graph

1. Carry out liveness analysis for the function
2. Create a graph node for each live range
3. for each basic block B, traverse B backwards:
   - let LR_x denote the live range for a given occurrence of a variable x
   - initialize LiveNow = $\{LR_w \mid w \in \text{LiveOut}(B)\}$
   - for each instruction $x = y \oplus z$:
     - for each live range $LR_i \in$ LiveNow :
       add the edge (LR_x, LR_i)
     - update:
       LiveNow = $(\text{LiveNow} - \{LR_x\}) \cup \{LR_y, LR_z\}$

Choosing a node to spill

- Estimating spill cost:
  - Let $Refs(x)$ = the set of points where a variable x is referenced (i.e., defined or used); and
  - $freq(p)$ = the execution frequency of a point p

  Then the cost of spilling x is (roughly):

  $$cost(x) = \Sigma\{\, freq(p) \mid p \in Refs(x) \,\}$$

- Spilling:
  - choose a node to minimize cost/degree. [Chaitin 82]

  This picks nodes that are relatively inexpensive to spill, but which lowers the degrees of many other nodes.

Estimating Execution Frequencies

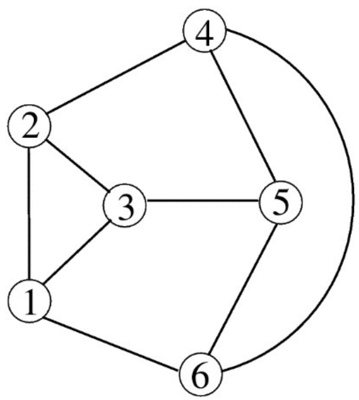
Simple approach: heuristics relating to loop nesting depth:

- preorder traversal of the syntax tree;
- execution frequency of root node = 1;
- each loop assumed to execute a “reasonable” no. of times (typically, 8–12);
- each branch of a conditional assumed to be equally likely.

This effectively pushes spill code away from inner loops.

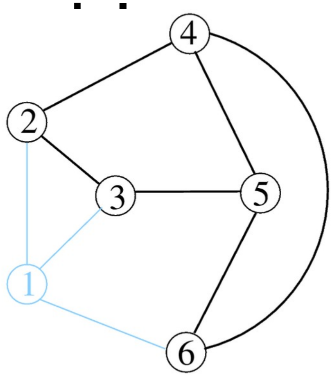
Spilling: Example

Interference graph (suppose the no. of colors = 3):



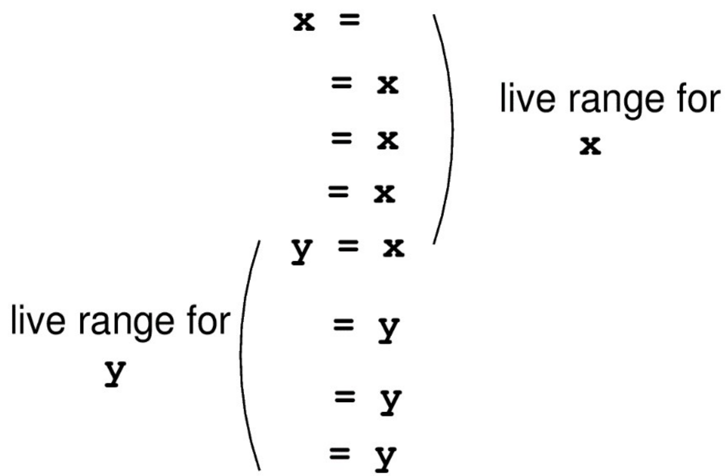
Node	*Cost*	*Cost/Degree*
1	2	0.67
2	11	3.67
3	21	7.00
4	5	1.67
5	5	1.67
6	3	1.00

After spilling node 1, the graph becomes 3-colo



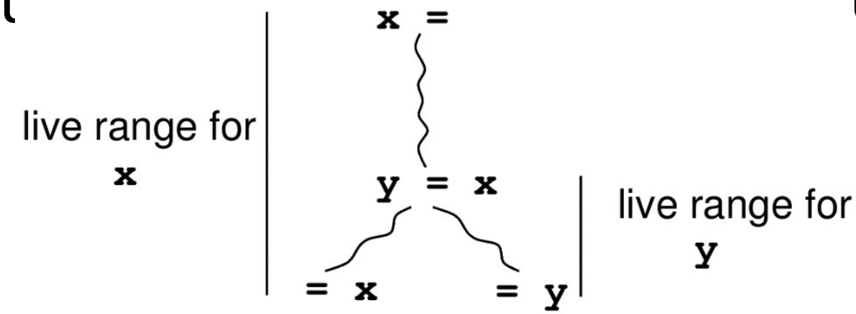
Coalescing Live Ranges

- Sometimes we can use one register for two different live ranges.
- Benefits:
  - Coalescing live ranges LR1, LR2 reduces the degree of any live range that interferes with both LR1 and LR2.
  - Eliminates the copy operation.
  - Reduces the no. of live ranges the compiler has to deal with.



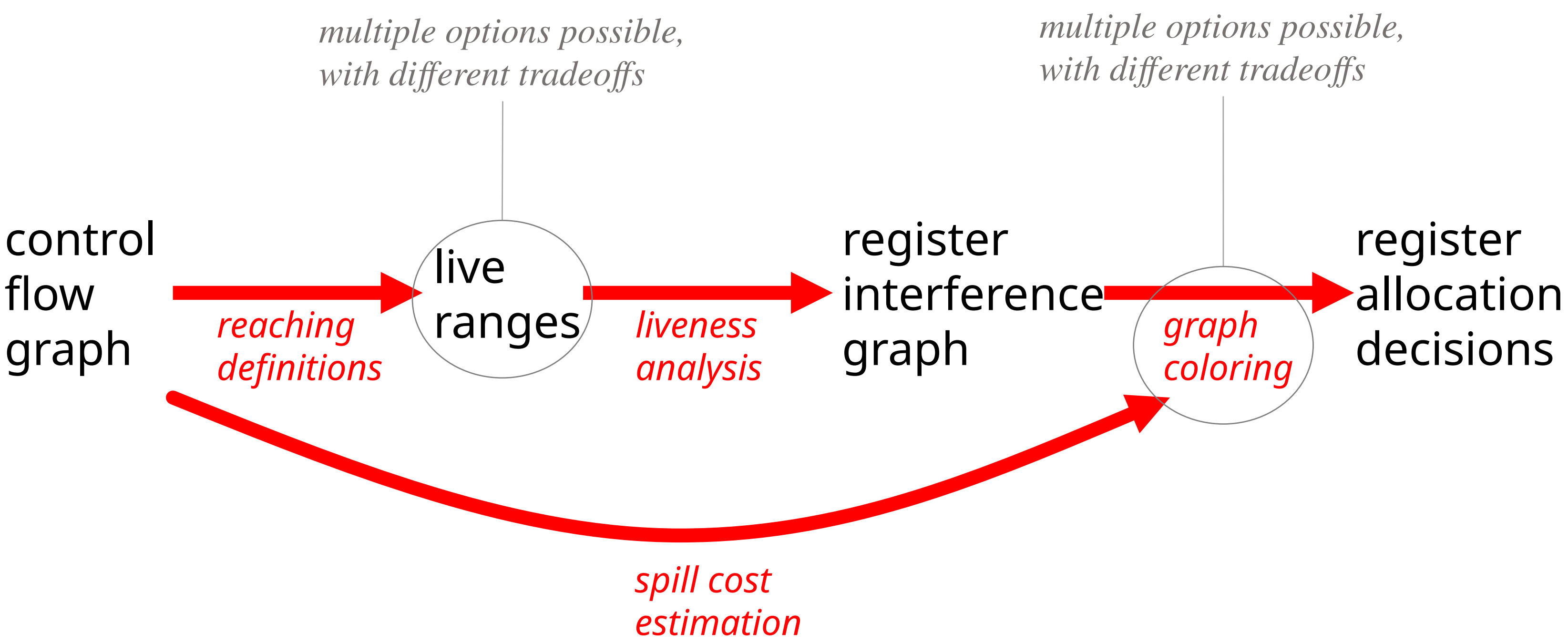
Coalescing Live Ranges (cont'd)

- In general, coalesce two live ranges A and B if:
  - A and B are connected only at a copy statement; and
  - don't interfere with each other elsewhere.



- Ordering of coalescing:
  - Coalescing two live ranges can prevent subsequent coalescing of other live ranges (i.e., ordering matters).
  - Consider coalescing for copy instructions with highest execution count first.

Register allocation: Summary



Register allocation: Overall Algorithm

1. Find live ranges, construct the interference graph.
2. repeat until no change:
   - coalesce live ranges to eliminate copy instructions
   - recompute interferences.
3. Estimate spill costs.
4. Simplify the interference graph
   - e.g., use heuristic discussed earlier
5. Use the stack to visit unspilled nodes in reverse order, and assign colors to them
6. Generate code to use assigned registers.

Code generation

Without register allocation:

```
codegen("x = y + z"):
    reg1 = gen_load(y)
    reg2 = gen_load(z)
    reg3 = find_reg()
    emit("reg3 := reg1 + reg2")
    gen_store(reg3, x)
```

Code generation

With~~out~~ register allocation:

```
codegen("x = y + z"):
    reg1 = gen_load(y, regs_to_use, regs_to_avoid)
    reg2 = gen_load(z, regs_to_use, regs_to_avoid)
    reg3 = find_reg(regs_to_use, regs_to_avoid)
    emit("reg3 := reg1 + reg2")
    gen_store(reg3, x)
```

Effects of Global Register Allocation

[Chow & Hennessey: MIPS C compiler]

Program	*% Reduction*		
	cycles	total loads/stores	scalar loads/stores
bm (theorem prover)	37.6	76.9	96.2
diff (file comparison utility)	40.6	69.4	92.5
yacc (parser generator)	31.2	67.9	84.4
nroff (document formatter)	16.3	49.0	54.7
C compiler front end	25.0	53.1	67.2
MIPS assembler	30.5	54.6	70.8