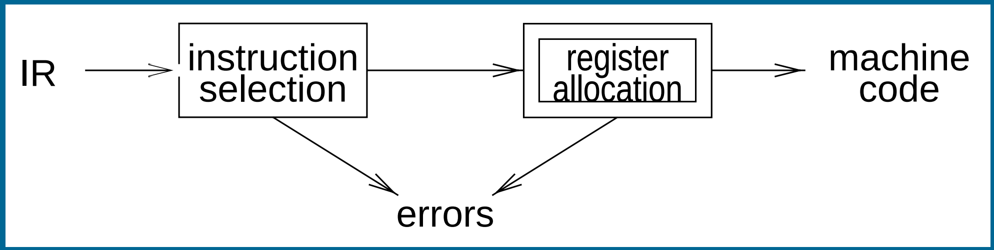
Register allocation



Register allocation:

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult
  $\Rightarrow$ NP-complete for $k \geq 1$ registers

Register allocation by simplification

1. *Build* interference graph G: for each program point
   (a) compute set of temporaries simultaneously live
   (b) add edge to graph for each pair in set
2. *Simplify*: Color graph using a simple heuristic
   (a) suppose G has node m with degree $< K$
   (b) if $G' = G - \{m\}$ can be colored then so can G, since nodes adjacent to m have at most $K-1$ colors
   (c) each such simplification will reduce degree of remaining nodes leading to more opportunity for simplification
   (d) leads to recursive coloring algorithm
3. *Spill*: suppose $\nexists m$ of degree $< K$
   (a) target some node (temporary) for spilling (optimistically, spilling node will allow coloring of remaining nodes)
   (b) remove and continue simplifying
4. *Select*: assign colors to nodes
   (a) start with empty graph
   (b) if adding non-spill node there must be a color for it as that was the basis for its removal
   (c) if adding a spill node and no color available (neighbors already K-colored) then mark as an *actual spill*

(d) repeat select

5. *Start over*: if select has no actual spills then finished, otherwise

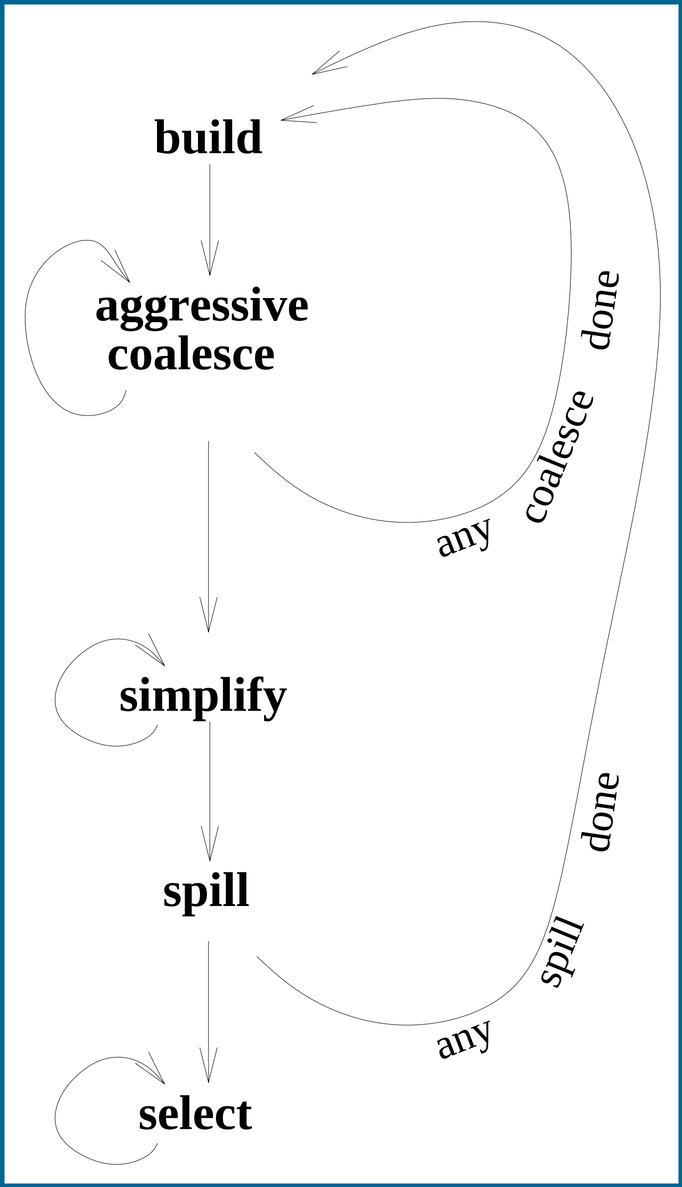
   (a) rewrite program to fetch actual spills before each use and store after each definition

   (b) recalculate liveness and repeat

Coalescing

- Can delete a *move* instruction when source s and destination d do not interfere:
  - *coalesce* them into a new node whose edges are the union of those of s and d
- In principle, any pair of non-interfering nodes can be coalesced
  - unfortunately, the union is more constrained and new graph may no longer be K-colorable
  - overly aggressive

Simplification with aggressive coalescing



Conservative coalescing

Apply tests for coalescing that preserve colorability.

Suppose a and b are candidates for coalescing into node ab

Briggs: coalesce only if ab has $< K$ neighbors of *significant* degree $\geq K$

- *simplify* will first remove all insignificant-degree neighbors
- ab will then be adjacent to $< K$ neighbors
- *simplify* can then remove ab

George: coalesce only if all significant-degree neighbors of a already interfere with b

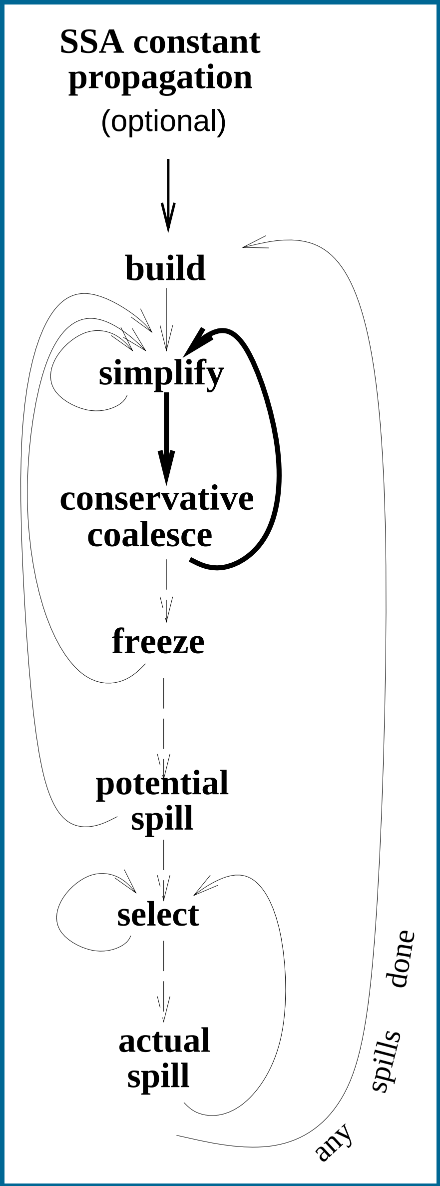
- *simplify* can remove all insignificant-degree neighbors of a
- remaining significant-degree neighbors of a already interfere with b so coalescing does not increase the degree of any node

Iterated register coalescing

Interleave simplification with coalescing to eliminate most moves while without extra spills

1. *Build* interference graph G; distinguish move-related from non-move-related nodes
2. *Simplify*: remove non-move-related nodes of low degree one at a time
3. *Coalesce*: conservatively coalesce move-related nodes
   - remove associated move instruction
   - if resulting node is non-move-related it can now be simplified
   - repeat simplify and coalesce until only significant-degree or uncoalesced moves
4. *Freeze*: if unable to simplify or coalesce
   (a) look for move-related node of low-degree
   (b) freeze its associated moves (give up hope of coalescing them)
   (c) now treat as a non-move-related and resume iteration of simplify and coalesce
5. *Spill*: if no low-degree nodes
   (a) select candidate for spilling
   (b) remove to stack and continue simplifying
6. *Select*: pop stack assigning colors (including actual spills)
7. *Start over*: if select has no actual spills then finished, otherwise
   (a) rewrite code to fetch actual spills before each use and store after each definition
   (b) recalculate liveness and repeat

Iterated register coalescing



Spilling

- Spills require repeating *build* and *simplify* on the whole program
- To avoid increasing number of spills in future rounds of *build* can simply discard coalescences
- Alternatively, preserve coalescences from before first *potential* spill, discard those after that point
- Move-related spilled temporaries can be aggressively coalesced, since (unlike registers) there is no limit on the number of stack-frame locations

Precolored nodes

Precolored nodes correspond to machine registers (e.g., stack pointer, arguments, return address, return value)

- *select* and *coalesce* can give an ordinary temporary the same color as a precolored register, if they don't interfere
- e.g., argument registers can be reused inside procedures for a temporary
- *simplify*, *freeze* and *spill* cannot be performed on them
- also, precolored nodes interfere with other precolored nodes

So, treat precolored nodes as having infinite degree

This also avoids needing to store large adjacency lists for precolored nodes; coalescing can use the George criterion

Temporary copies of machine registers

Since precolored nodes don't spill, their live ranges must be kept short:

1. use *move* instructions
2. move callee-save registers to fresh temporaries on procedure entry, and back on exit, spilling between as necessary
3. *register pressure* will spill the fresh temporaries as necessary, otherwise they can be coalesced with their precolored counterpart and the moves deleted

Caller-save and callee-save registers

Variables whose live ranges span calls should go to callee-save registers, otherwise to caller-save

This is easy for graph coloring allocation with spilling

- calls interfere with caller-save registers
- a cross-call variable interferes with all precolored caller-save registers, as well as with the fresh temporaries created for callee-save copies, forcing a spill
- choose nodes with high degree but few uses, to spill the fresh callee-save temporary instead of the cross-call variable
- this makes the original callee-save register available for coloring the cross-call variable

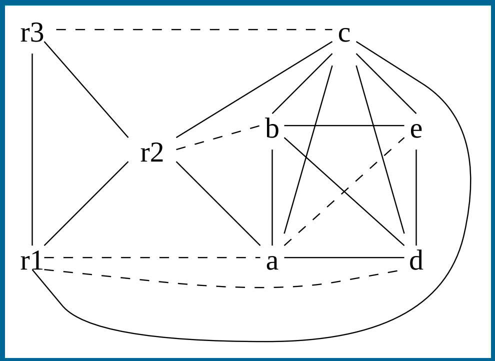
Example

```
enter:
  c := r3
  a := r1
  b := r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 goto loop
  r1 := d
  r3 := c
  return [ r1, r3 live out ]
```

- Temporaries are `a`, `b`, `c`, `d`, `e`
- Assume target machine with $K = 3$ registers: `r1`, `r2` (caller-save/argument/result), `r3` (callee-save)
- The code generator has already made arrangements to save `r3` explicitly by copying into temporary `a` and back again

Example (cont.)

- Interference graph:



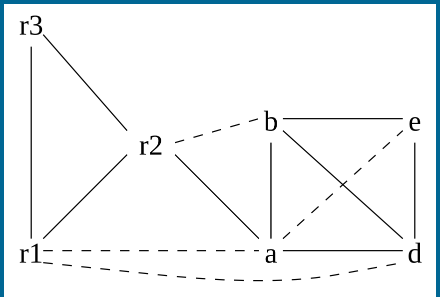
- No opportunity for *simplify* or *freeze* (all non-precolored nodes have significant degree $\geq K$)
- Any *coalesce* will produce a new node adjacent to $\geq K$ significant-degree nodes
- Must *spill* based on priorities:

Node		uses + defs outside loop		uses + defs inside loop		degree		priority
`a`	(	2	$+10\times$	0	)/	4	=	0.50
`b`	(	1	$+10\times$	1	)/	4	=	2.75
`c`	(	2	$+10\times$	0	)/	6	=	0.33
`d`	(	2	$+10\times$	2	)/	4	=	5.50
`e`	(	1	$+10\times$	3	)/	3	=	10.30

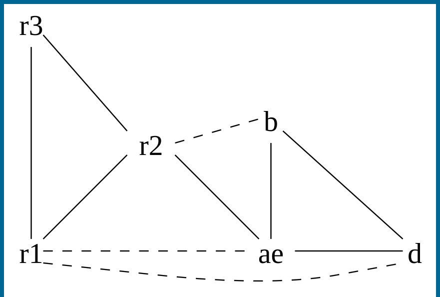
- Node `c` has lowest priority so spill it

Example (cont.)

- Interference graph with `c` removed:

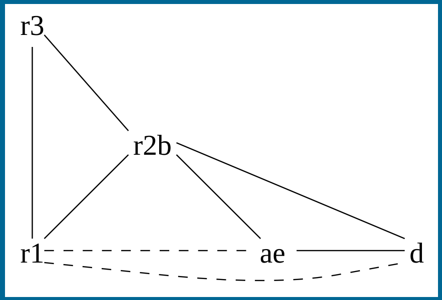


- Only possibility is to *coalesce* `a` and `e`: `ae` will have $< K$ significant-degree neighbors (after coalescing `d` will be low-degree, though high-degree before)

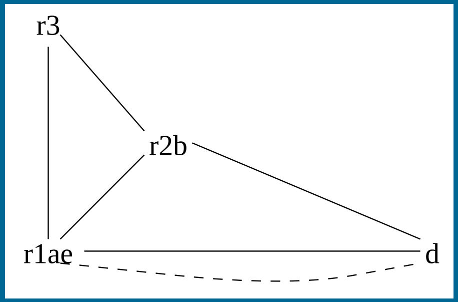


Example (cont.)

- Can now *coalesce* `b` with `r2` (or coalesce `ae` and `r1`):

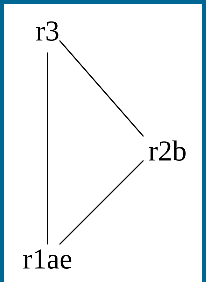


- *Coalescing* `ae` and `r1` (could also coalesce `d` with `r1`):



Example (cont.)

- Cannot *coalesce* `r1ae` with `d` because the move is *constrained*: the nodes interfere. Must *simplify* `d`:



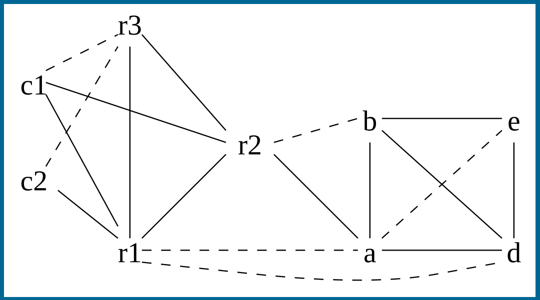
- Graph now has only precolored nodes, so pop nodes from stack coloring along the way
  - `d` $\equiv$ `r3`
  - `a`, `b`, `e` have colors by coalescing
  - `c` must spill since no color can be found for it
- Introduce new temporaries `c1` and `c2` for each use/def, add loads before each use and stores after each def

Example (cont.)

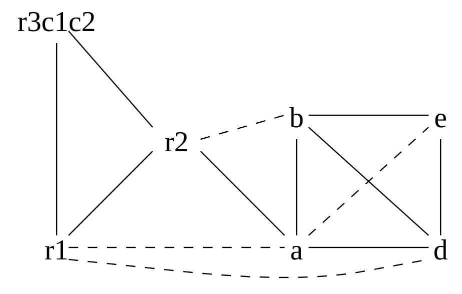
```
enter:
  c1 := r3
  M[c_loc] := c1
  a := r1
  b := r2
  d := 0
  e := a
loop:
  d := d + b
  e := e - 1
  if e > 0 goto loop
  r1 := d
  c2 := M[c_loc]
  r3 := c2
  return [ r1, r3 live out ]
```

Example (cont.)

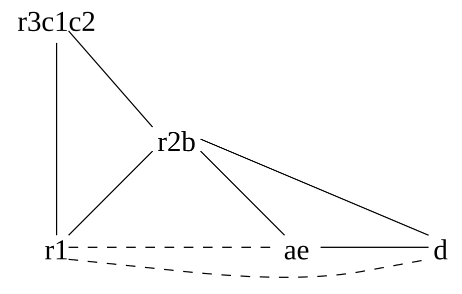
- New interference graph:



- *Coalesce* `c1` with `r3`, then `c2` with `r3`:

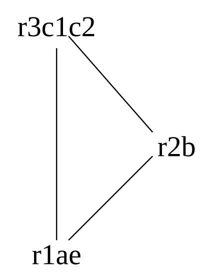


- As before, *coalesce* `a` with `e`, then `b` with `r2`:



Example (cont.)

- As before, *coalesce* `ae` with `r1` and *simplify* `d`:



- Pop `d` from stack: select `r3`. All other nodes were coalesced or precolored. So, the coloring is:
  - $\mathtt{a} \equiv \mathtt{r1}$
  - $\mathtt{b} \equiv \mathtt{r2}$
  - $\mathtt{c} \equiv \mathtt{r3}$
  - $\mathtt{d} \equiv \mathtt{r3}$
  - $\mathtt{e} \equiv \mathtt{r1}$

Example (cont.)

- Rewrite the program with this assignment:

```
enter:
  r3 := r3
  M[c_loc] := r3
  r1 := r1
  r2 := r2
  r3 := 0
  r1 := r1
loop:
  r2 := r3 + r2
  r1 := r1 - 1
  if r1 > 0 goto loop
  r1 := r3
  r3 := M[c_loc]
  r3 := r3
  return [ r1, r3 live out ]
```

Example (cont.)

- Delete moves with source and destination the same (coalesced):

```
enter:
  M[c_loc] := r3
  r3 := 0
loop:
  r2 := r3 + r2
  r1 := r1 - 1
  if r1 > 0 goto loop
  r1 := r3
  r3 := M[c_loc]
  return [ r1, r3 live out ]
```

- One uncoalesced move remains