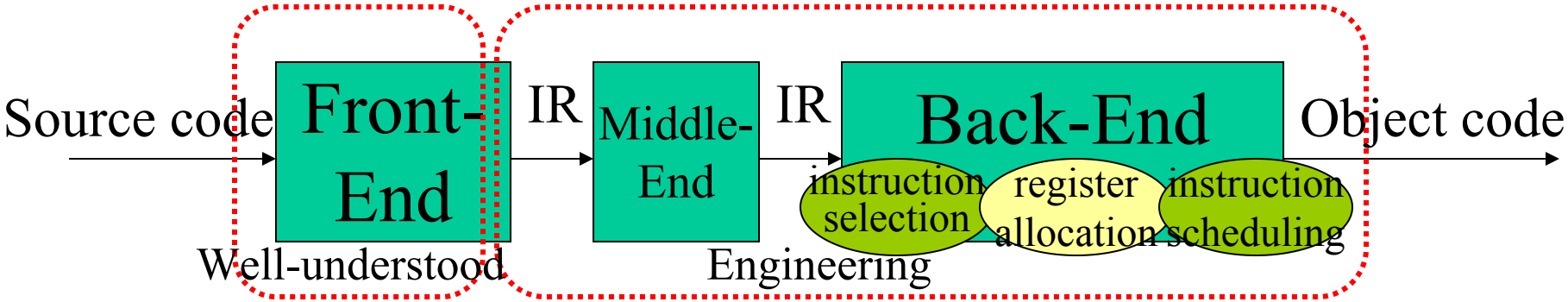
Lecture 17: Register Allocation via Graph Colouring



- Last Lecture: (Local) Register Allocation.
- Global Register Allocation:
  - Go beyond basic blocks.
  - The task (again!):
    - Produce correct *k* register code.
    - Minimise number of loads and stores (spill code) and their space.
    - The allocator must be efficient (e.g., no backtracking)
- Today: Register Allocation via Graph Coloring:
    - 1st part: within a basic block. 2nd part: globally.

Register Allocation via graph colouring

The idea:

- live ranges that do not interfere can share the same register.

The algorithm:

1. Construct live ranges
2. Build ***interference graph***
3. (try to) construct a ***k-colouring*** of the graph:
   - If unsuccessful, choose values to spill (on the basis of some cost estimates in each case) and repeat from start (spill placement becomes a critical issue).
4. Map colours onto physical registers

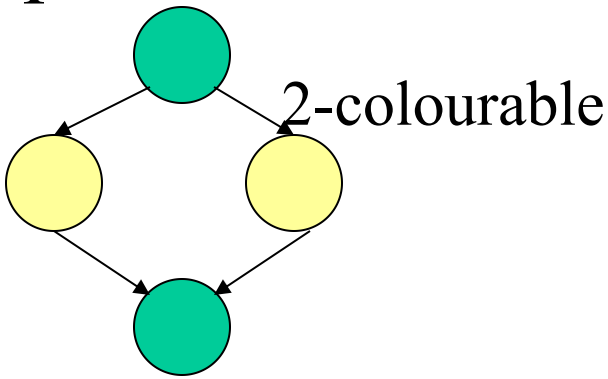
(can be generalised for global register allocation, by constructing global live ranges and building interference graph for procedure)

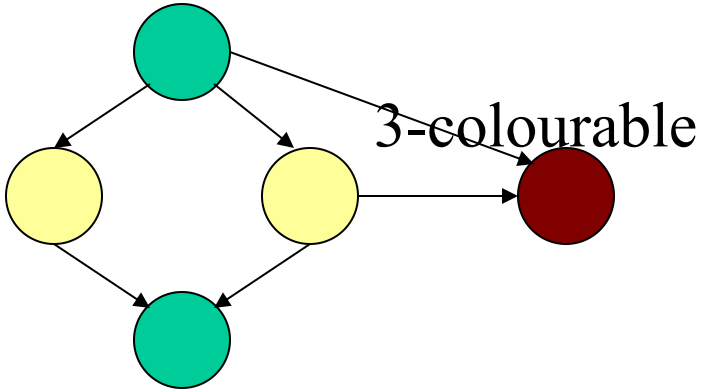
Graph Colouring - Background

- The problem:

  A graph is said to be *k*-colourable iff the nodes can be labelled with integers *1*…*k* so that no edge connects nodes with the same label.

- Examples:





A colouring that uses *k* colours is termed a *k*-colouring and *k* is the graph's *chromatic number*.

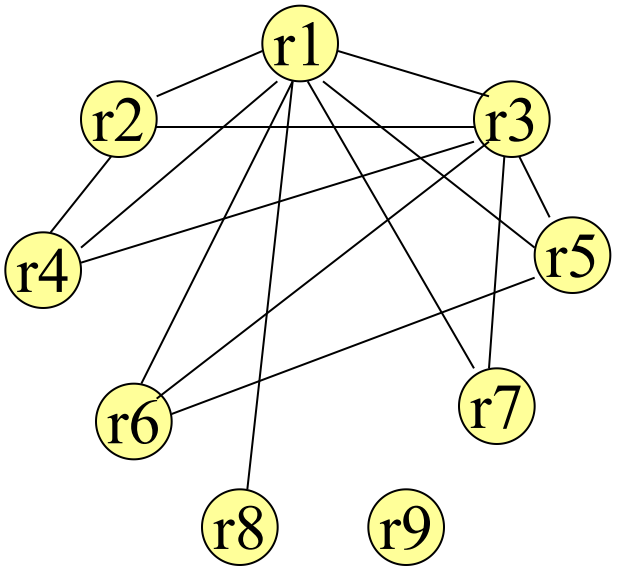
(Famous problem: the map-colouring problem)

Build the interference graph

- What is an interference?
  - Two values interfere if there exists an operation where both are simultaneously live; if they interfere they cannot occupy the same register.
- The interference graph:
  - Nodes: represent values (or live ranges).
  - Edges: represent individual interferences.

Example (using the basic block from last lecture):

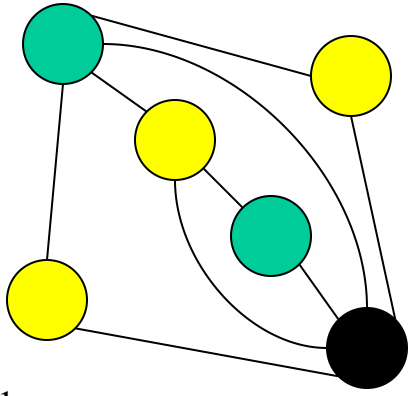
r1	[1,9]	*	*	*	*	*	*	*	*	*
r2	[2,5]		*	*	*	*				
r3	[3,8]			*	*	*	*	*	*	
r4	[4,5]				*	*				
r5	[5,7]					*	*	*		
r6	[6,7]						*	*		
r7	[7,8]							*	*	
r8	[8,9]								*	*
r9	[9,9]									*
# of live values		1	2	3	4	4	4	4	3	2

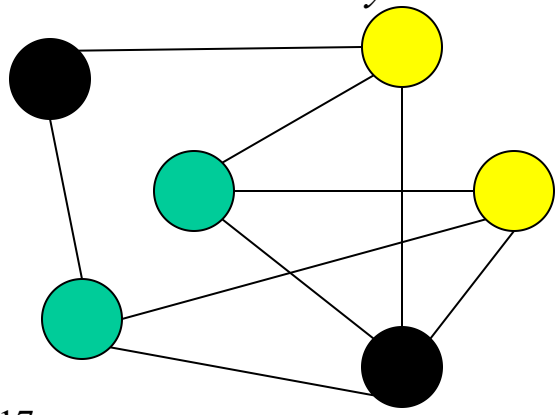


Construct a k-colouring

- **Top-down colouring:**
  - Rank the live ranges (that is, the nodes):
    - Possible ways of ranking: number of neighbours (in decreasing order), spill cost (starting with nodes that is more important to have in registers)
  - Follow the ranking to assign colours:
    - (for each node, pick the first colour that is not used by the node's neighbours)
  - If a live range cannot be coloured: spill (store after definition, load before each use) or split the live range.

(Observation: every node with a number of neighbours less than k will always receive a colour!)



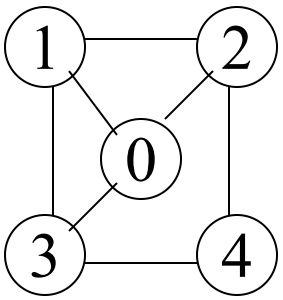


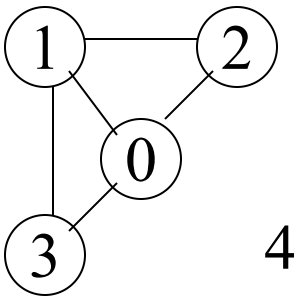
Construct a k-colouring (cont.)

- **Bottom-up colouring**: (Chaitin's algorithm)
  - 1. Simplify the graph:
    - Pick any node, such that the number of neighbouring nodes is less than k (degree of the node), and put it on the stack.
    - Remove that node and all edges incident to it
  - 2. If the graph is non-empty (i.e., all nodes have k or more neighbours), then:
    - Use some heuristic to spill a live range; remove corresponding node; if this causes some neighbours to have fewer than k nodes goto step 1, otherwise repeat step 2.
  - 3. Successively pop nodes off the stack and colour them using the first colour not used by some neighbour.
    - If a node cannot be coloured, leave it uncoloured (will have to spill).

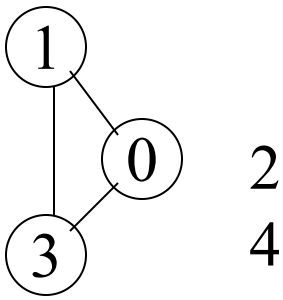
(Observation: A graph having a node n with degree < k is k-colourable iff the graph with node n removed is k-colourable)

Bottom-up colouring: example (k=3)

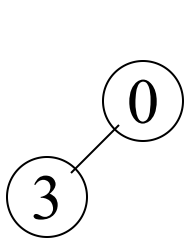




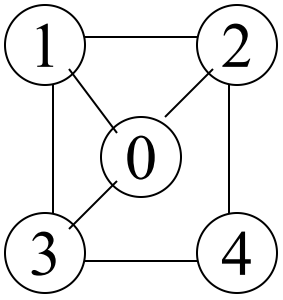
4



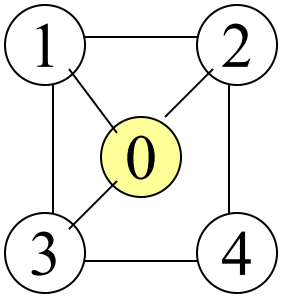
2
4



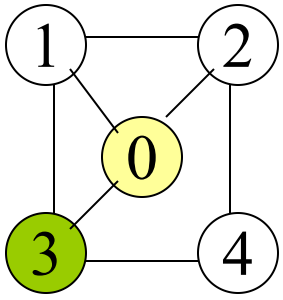
1
2
4



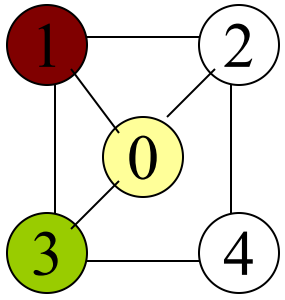
3
1
2
4



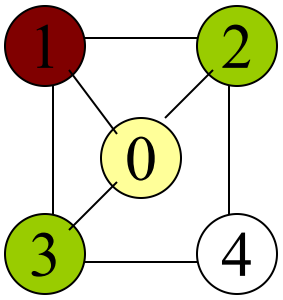
3
1
2
4



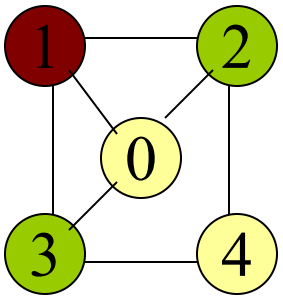
1
2
4



2
4

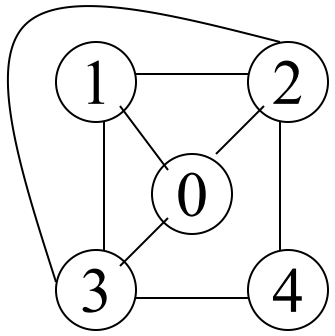


4



What about

?



Global Register Allocation via graph colouring

The idea:

- live ranges that do not interfere can share the same register.

The algorithm:

1. Construct **global** live ranges
2. Build ***interference graph*** for **procedure**
3. (try to) construct a ***k-colouring*** of the graph:
   - If unsuccessful, choose values to spill (on the basis of some cost estimates in each case) and repeat from start (spill placement becomes a critical issue).
4. Map colours onto physical registers

(we need to deal with 1 and 2)

Global live ranges

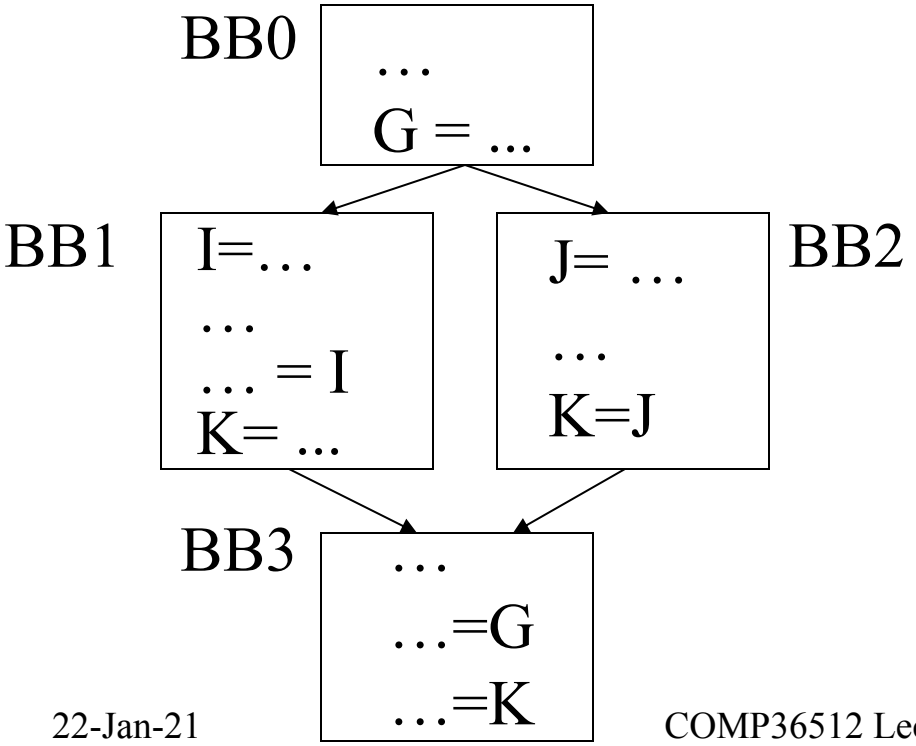
- At some point p in a procedure, a value v is live if it has been defined along a path from the procedure entry's to p and a path exists from p to a use of v, along which v is not redefined.

  (Hmm, paths imply that we need the Control Flow Graph!)

- To discover global live ranges, the compiler must discover the set of entries that are live on entry to each basic block, as well as those that are live on exit from each basic block. Each basic block, b, is annotated with:
  - LIVEIN(b): a value is in LIVEIN if it is defined along some path through the control-flow graph that leads to b and it is either used directly in b or it is in LIVEOUT(b).
  - LIVEOUT(b): a value is in LIVEOUT if it is used along some path leaving b before being redefined, and it is either defined in b or is in LIVEIN(b).

Construct LIVEIN, LIVEOUT

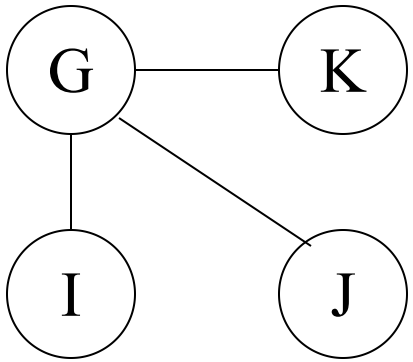
- If basic block has no successors, LIVEOUT(b) = ∅
- For all other basic blocks: LIVEOUT(b)=∪ LIVEIN(s) for all immediate successors, s, of b in the control flow graph.
- A value is in LIVEIN(b):
  - if it is used before it is defined (if it is defined) in basic block b; or
  - it is not used, nor defined, but it is in LIVEOUT(b)



LIVEOUT(BB3) ={}
LIVEIN(BB3)={G,K}
LIVEOUT(BB2)={G,K}
LIVEIN(BB2)={G}
LIVEOUT(BB1)={G,K}
LIVEIN(BB1)={G}
LIVEOUT(BB0}={G}
LIVEIN(BB0)={}

Build the interference graph

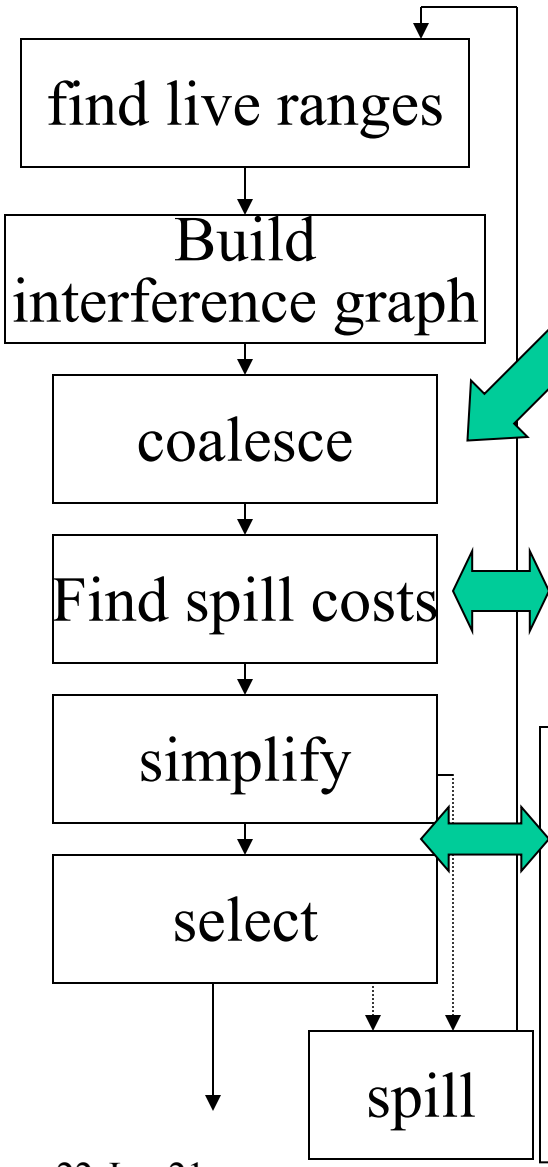
for each basic block b
    LIVENOW(b)←LIVEOUT(b)
    for each operation in b (in reverse order) of the type op lr1,lr2,lr3
        for each live_range in LIVENOW(b) except lr2 and lr3
            add an edge between live_range and lr1
        remove lr1 from LIVENOW(b)
        add lr2 and lr3 to LIVENOW(b)



Estimate spill costs

- Components of cost for a spill:
  - Address computation:
    - Minimise by keeping spilled values in activation record.
    - Perform load/store with (pointer+offset) instruction.
  - Memory operation:
    - Unavoidable (compiler hopes that spill locations stay in the cache).
  - Estimated execution frequencies:
    - Static analysis to estimate execution counts for basic blocks: spill in outer loops not in inner loops.

Chaitin-Briggs Register Allocators



Coalesce: Consider the operation mov r1, r2. If r1, r2 don't interfere, the operation can be eliminated and all references to r1 can be rewritten to use r2.

Chaitin's heuristic: use max(spill_cost/degree). degree is the number of neighbours and spill cost is (#refs)*10^d–(# LD+STORE)*2*10^d, where d is the loop nesting depth.

Simplify will push onto the stack nodes with less than k neighbours. If at some point only nodes with ≥ k neighbours exist, Chaitin stops and spills. Briggs will continue, and will do the spilling, during the coloring phase (when pop-ping from the stack) if no color can be assigned.

Exercise: Perform Register Allocation using Graph Colouring

1. mov r1,1
2. shl r2, r1, r1
3. shl r3, r2, r1
4. add r20, r1, r2
5. shl r4, r3, r1
6. shl r5, r4, r1
7. add r21, r3, r4
8. add r30, r20, r21
9. shl r6, r5, r1
10. add r22, r5, r6
11. shl r7, r6, r1
12. shl r8, r7, r1
13. add r23, r7, r8
14. add r31, r22,r23
15. shl r9, r8, r1
16. shl r10, r9, r1
17. shl r11, r10, r1
18. add r24, r9, r10
19. add r35, r24, r31
20. store r35

Conclusion

- Register allocation is an active research field. Main problem: minimise spill costs.
- Register allocation through graph colouring is popular because colouring captures critical aspects of the global problem.
- Size of interference graph may be quite large.
- Huge amount of work in the literature and an active research field:
  - Different solutions tend to be sensitive to small decisions.
  - Variations address compile-time speed/quality of the resulting code.
- For those interested:
  - G. Chaitin *et al*. "Register Allocation via Coloring". Computer Languages 6(1), Jan. 1981.
  - P. Briggs *et al*. "Improvements to Graph Coloring Register Allocation". ACM Transactions on Programming Languages and Systems, 16(3), May 1994 (and PLDI 1989).
  - Traub *et al*. "Quality and Speed in Linear-scan Register Allocation". ACM SIGPLAN'98 PLDI, 1998.
- Reading: Aho2, pp.556-557; Aho1, pp 545-546; Hunter, pp.202-204 (all of these are too condensed); Cooper, Chapter 13.
- Next time: Instruction Scheduling