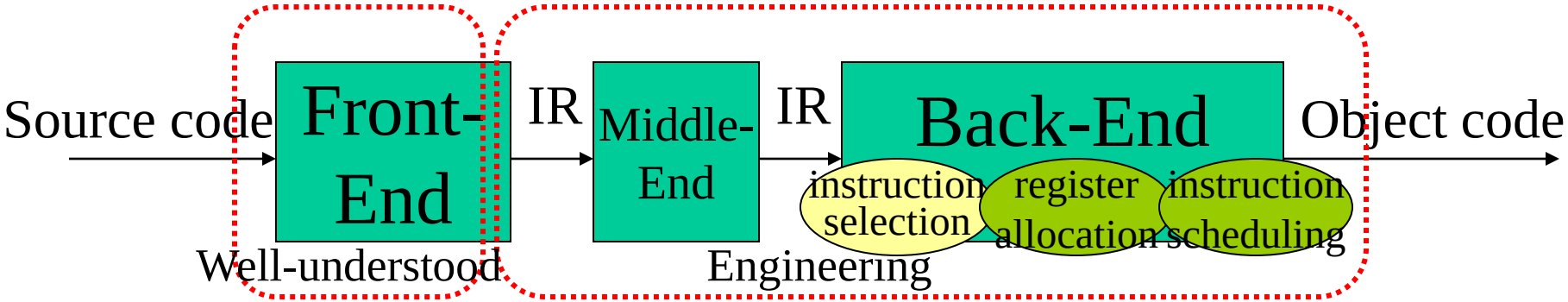
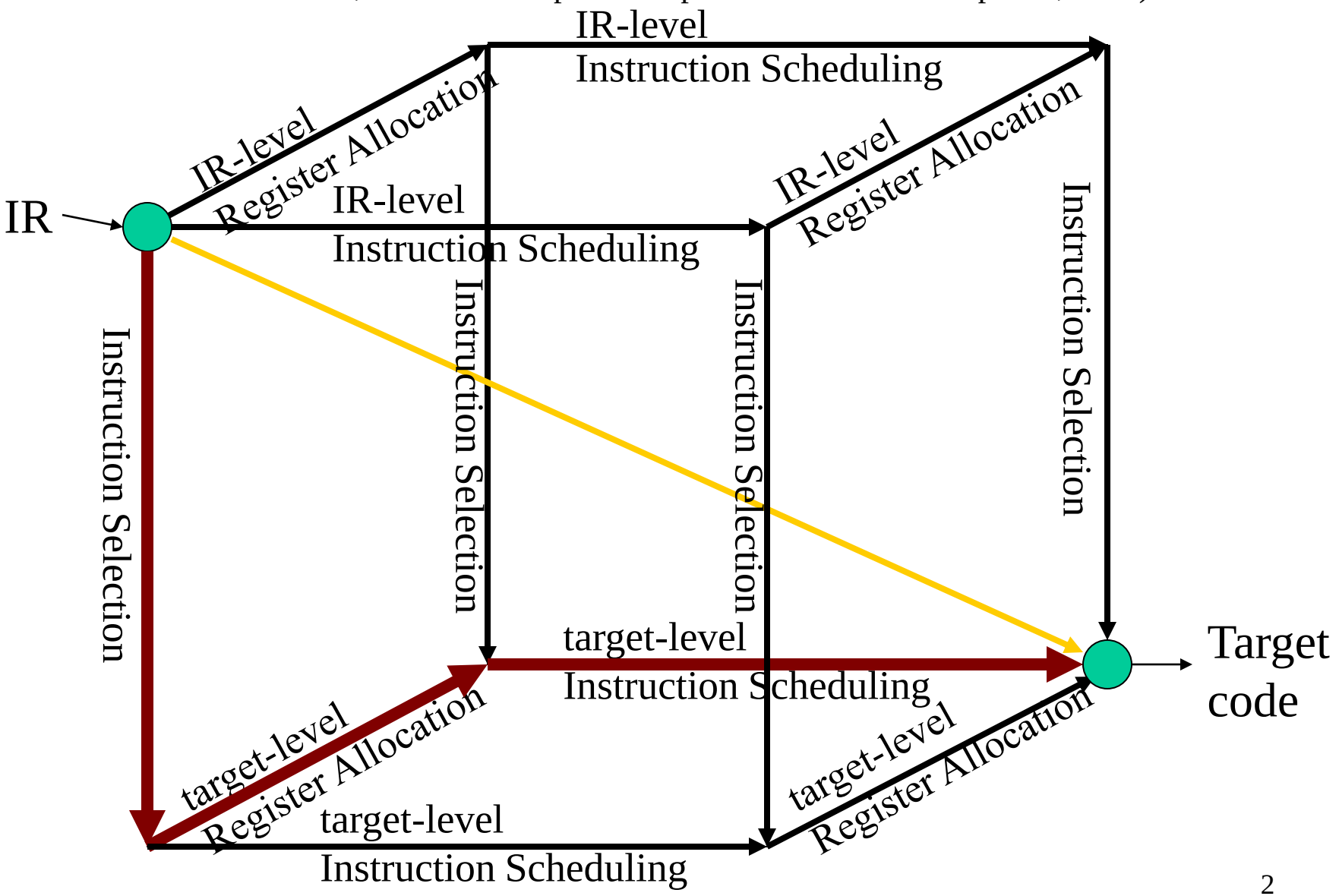
Register Allocation and Instruction Scheduling



- Background:
  - Code generation: we assume there are 3 key tasks that should be performed:
    - instruction selection: mapping intermediate representation (IR) into assembly code (mostly a pattern matching problem)
    - register allocation: decide which values will reside in registers.
    - instruction scheduling: reorder operations to hide latencies…
  - Conventional wisdom says that we lose little by solving these (NP-complete) problems independently.
  - The focus of these two lectures will be register allocation and instruction scheduling.

Code Generation as a phase-decoupled problem

(thanks for inspiration to: C.Kessler, A.Bednarski; Optimal Integrated Code Generation for VLIW architectures; 10th Workshop on Compilers for Parallel Computers, 2003)



Instruction Selection

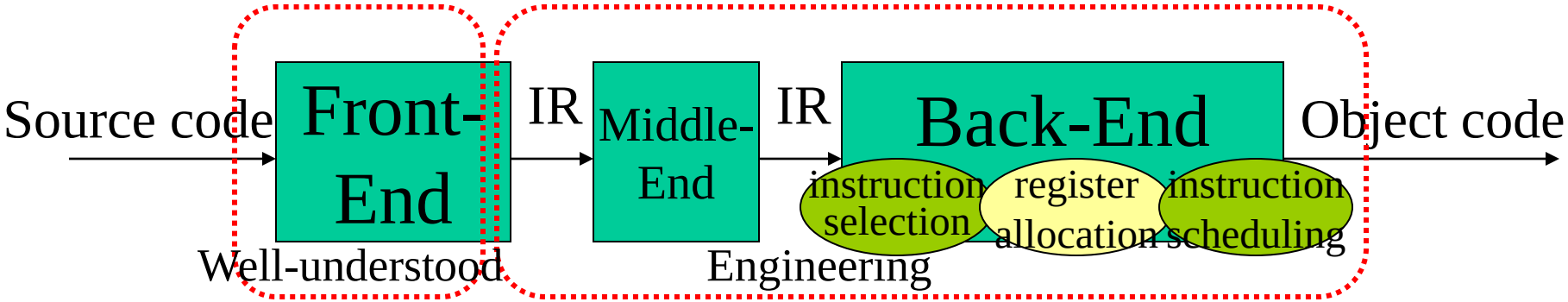
- Characteristics:
  - use some form of pattern matching
  - can make locally optimal choices;global optimality is NP-complete
  - assume enough registers.
- Assume a RISC-like target language
- Recall:
  - modern processors can issue multiple instructions at the same time.
  - instructions may have different **latencies**: e.g., add: 1 cycle; load 1-3 cycles; imult: 3-16 cycles, etc
- From a compiler point of view we are interested in:
  - instruction latency: length time elapsed from the time the instruction started execution until the time the results can be used.

Our (very basic) instruction set

- **load r1, @a** ; load register r1 with the memory value for a
- **load r1, [r2]** ; load register r1 with the memory value pointed by r2
- **mov r1, 3** ; r1=3
- **add r1, r2, r3** ; r1=r2+r3

  (same for mult, sub, div)
- **shr r1, r1, 1** ; r1=r1>>1 (shift right; r1/2)
- **store r1** ; store r1 in memory
- **nop** ; no operation

Register Allocation (Part I)



- Last Lecture: Instruction Selection.
- Register Allocation:
  - We assume a RISC-like (three-address) type of code.
  - The code makes use of an unbounded number of registers (**virtual registers**) but the machine has only a limited number of registers (**physical registers**), say *k*.
  - The task:
    - 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)

Background

- **Basic Block**: a maximal length segment of straight-line (i.e., branch-free) code. (Importance: strongest facts are provable for branch-free code; problems are simpler; strongest techniques.)
- **Local Register Allocation**: within a single basic block.
- **Global Register Allocation**: across an entire procedure (multiple BBs).
- **Allocation**: choose what to keep in registers.
- **Assignment**: choose specific registers for values.
- Modern processors may have multiple register classes:
  - General-purpose, floating-point, branch target, …
  - Problem: interactions between classes - Assume separate allocation for each class.
- **Complexity**: Only simplified cases of local allocation and assignment can be solved in linear time. All the rest (including global allocation – even for 1 register – and most sub-problems) are NP-complete. We need good heuristics!

Real compilers face real problems!

Liveness and Live Ranges

- Problem: What is the number of registers needed in a basic block?
  - Naïve: all occurrences of a variable to the same register.
  - Realistic: Compute a set of live ranges and use their name space.
- A value of a variable is **live** between its definition and its uses:
  - Find definitions (x←…) and uses (…←…x…)
  - From definition to last use is the “live range”
  - Can represent live range as an interval [i,j] in basic block.
- Over all instructions in the basic block, let:
  - MAXLIVE be the maximum number of values live at an instruction
  - k, the number of physical registers available.
    - If MAXLIVE ≤ k, allocation is trivial.
    - If MAXLIVE > k, some values must be spilled to memory.

Example / Exercise

Compute live ranges for all registers and MAXLIVE in the following Basic Blocks:

```
1. load r1,@a
2. load r2,2
3. load r3,@b
4. load r4,@c
5. load r5,@d
6. mult r1,r1,r2
7. mult r1,r1,r3
8. mult r1,r1,r4
9. mult r1,r1,r5
10.store r1
```

		1	2	3	4	5	6	7	8	9	10
r1											
r1											
r1											
r1											
r1											
r2											
r3											
r4											
r5											
# of live values											

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. load r4,@x
5. sub  r5,r4,r2
6. load r6,@z
7. mult r7,r5,r6
8. sub  r8,r7,r3
9. add  r9,r8,r1
```

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

Top-Down (Local) Allocation

- Allocator must reserve *f* registers to ensure feasibility (e.g., for use in computations that involve values allocated to memory; 2 to 4 depending on the target processor).
- Idea (frequency count algorithm): keep *k–f* most frequently used values in the BB in a register; use *f* for the rest:
  - 1. Count number of uses for each virtual register.
  - 2. Assign top *k-f* virtual registers to physical registers.
  - 3. Rewrite code: if a virtual register was assigned to a physical register, replace. Else spill: use reserved registers to load before use and store after definition.
- Weakness: a value heavily used in the 1st half of the basic block and unused in the 2nd half, essentially wastes the register for the latter.

Example

- Assume 3 physical registers – two needed for feasibility.

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. load r4,@x
5. sub  r5,r4,r2
6. load r6,@z
7. mult r7,r5,r6
8. sub  r8,r7,r3
9. add  r9,r8,r1
```

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

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. store r3 //spill r3
5. load r3,@x
6. sub  r3,r3,r2
7. load r2,@z
8. mult r3,r3,r2
9. load r2, … //spilled value
10.sub  r3,r3,r2
11.add  r3,r3,r1
```

r1 is assigned to the most commonly used value (ok, there is a tie; we choose the first one), and r2 and r3 are used for feasibility!

This assumes that the compiler can realize that **`y`** is already in register **`r2`**, hence it is not necessary to do a **`load r2,@y`** again!

Bottom-Up (Local) Allocation

- Let multiple values occupy a single register – Best's algorithm:

  for each operation, i, 1 to N (op vr3, vr2, vr1)

  ensure that vr1 is in r1

  ensure that vr2 is in r2

  if r1 not needed after i, free(r1)

  if r2 not needed after i, free(r2)

  allocate r3 for vr3

  emit code – op r3,r2,r1

- **ensure**: if a vr is not in a physical register, **allocate** register and make sure that occurrences of vr are tied to this physical register.
- **allocate**: return a free physical register, or select the register that is used farthest in the future, store its value and return it.
- Due to Sheldon Best (1955) – often reinvented. Many have argued for its optimality…
- What does it remind you?

Example

- Assume 3 physical registers

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. load r4,@x
5. sub  r5,r4,r2
6. load r6,@z
7. mult r7,r5,r6
8. sub  r8,r7,r3
9. add  r9,r8,r1
```

By spilling the value here, note that MAXLIVE ≤ 3

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

```
1. load r1,@a
2. load r2,@y
3. mult r3,r1,r2
4. store r1 // spill the one used farthest
5. load r1,@x
6. sub  r1,r1,r2
7. load r2,@z
8. mult r1,r1,r2
9. sub  r1,r1,r3
10.load r2, … // load spilled value (load r2,@a)
11.add  r1,r1,r2
```

A ‘clever’ compiler may recognise that the store may not be needed since the value may be available from memory location @a (needs to guarantee that the value of this location won’t change)

More complex scenarios

- Basic blocks (BB) rarely exist in isolation:

  BB1: … store r17, @a is followed by

  BB2: load r12, @a …

  – Could replace load with a move; needs control-flow graph.
- Blocks with multiple (control-flow) predecessors:

  BB1: … store r4,@x and BB2: … store r7,@x followed by

  BB3: load r1, @x

  – What if BB1 has x in a register but BB2 not? (BB3 follows)
- Multiple basic blocks increase complexity:
  – How to compute the “farthest” in Best’s algorithm?

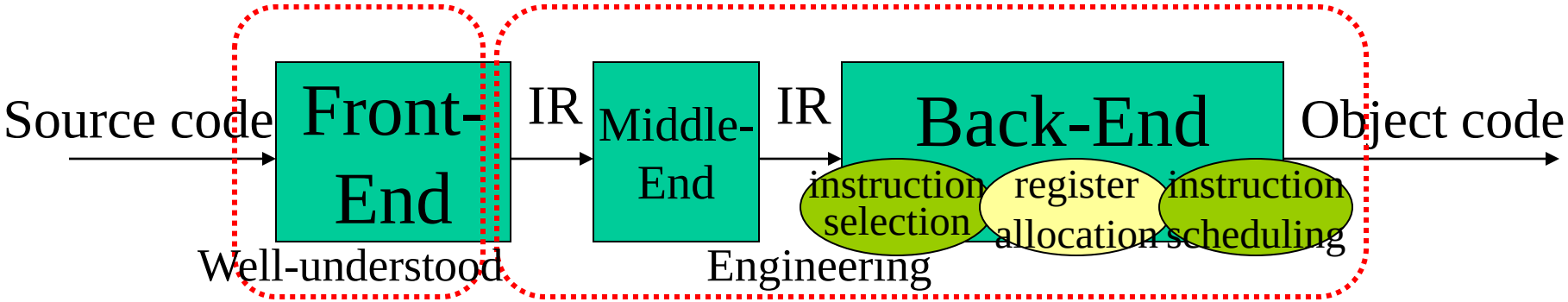
Conclusion (end of Part I)

- Register allocation in real cases is NP-complete.
- Best's algorithm, which has been reinvented repeatedly, performs well for local register allocation.

- Reading:
  - Aho2, pp. 553-556; Aho1, pp.541-546 (too condensed)
  - Cooper, Sections 13.1-13.4.1.

- Next part: Register allocation via graph colouring.

Register Allocation (Part II)



- Part I: (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)
- This part: Register Allocation via Graph Coloring:
    - 1st part: within a basic block. 2nd part: globally.

Global Register Allocation

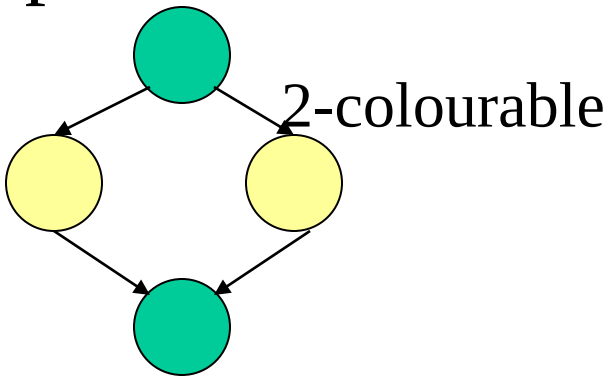
- Taking a global approach:
  - Abandon the distinction between local and global.
  - Generalised frequency counts: weigh uses and defs and BBs; apply to each BB; try to remove loads and stores between adjacent BBs (Fortran H; IBM 360,370)
- Graph colouring paradigm:
  - Build an interference graph:
  - (try to) construct a k-colouring
    - Minimal colouring is NP-complete
    - Spill placement becomes a critical issue
  - Map colours onto physical registers.

Graph Colouring - Background

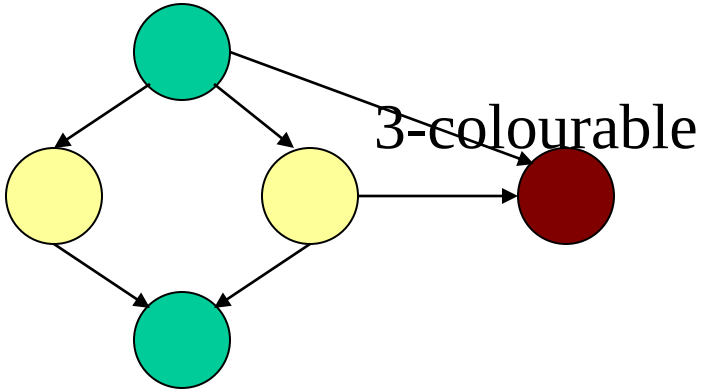
- The problem:

  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:



2-colourable



3-colourable

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)

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

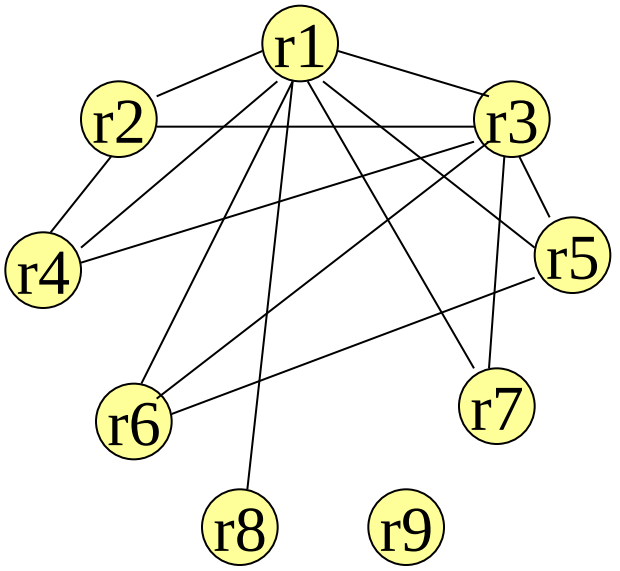
(can be generalised for global register allocation, by constructing global live ranges and building interference graph for procedure)

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):

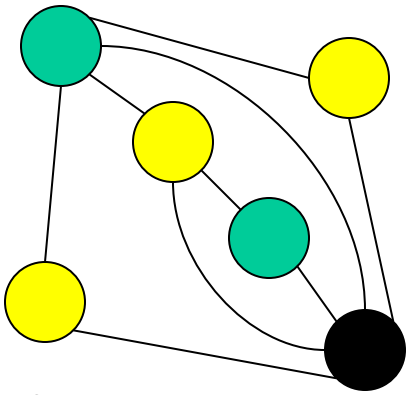
r1	[1,9]	*	*	*	*	*	*	*	*	*
r2	[2,5]		*	*	*	*				
r3	[3,8]			*	*	*	*	*	*	
r4	[4,5]				*	*				
r5	[5,7]					*	*	*		
r6	[6,7]						*	*		
r7	[7,8]							*	*	
r8	[8,9]								*	*
r9	[9,9]									*
# of live values		1	2	3	4	4	4	4	3	2

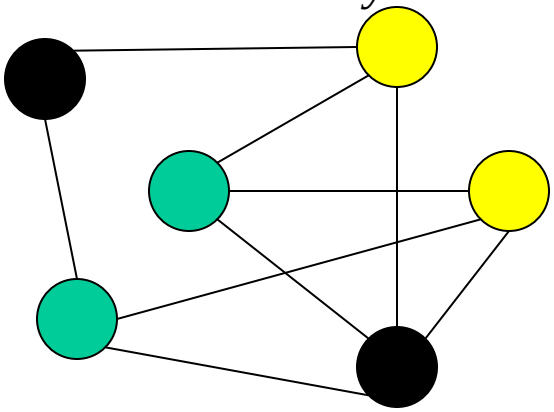


Construct a k-colouring

- **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!)



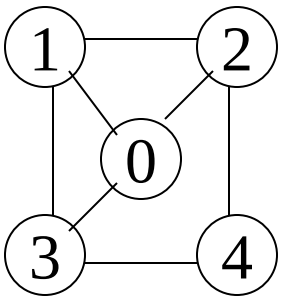


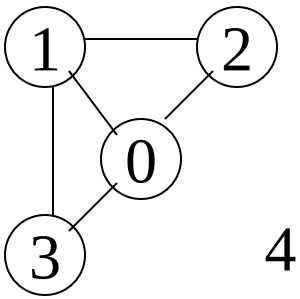
Construct a k-colouring (cont.)

- **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).

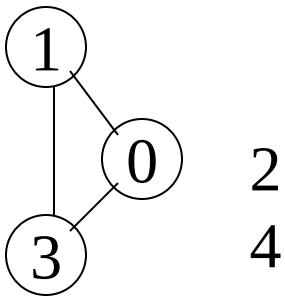
(Observation: A graph having a node n with degree < k is k-colourable iff the graph with node n removed is k-colourable)

Bottom-up colouring: example (k=3)

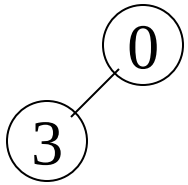




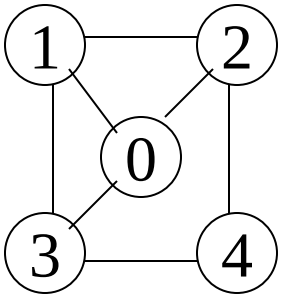
4



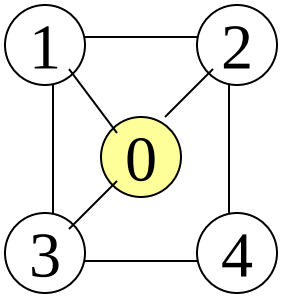
2
4



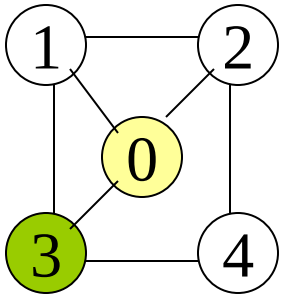
1
2
4



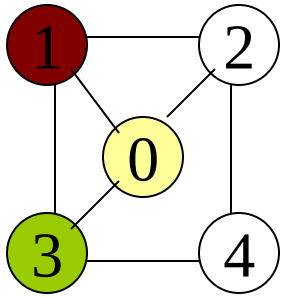
3
1
2
4



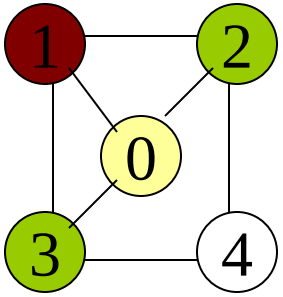
3
1
2
4



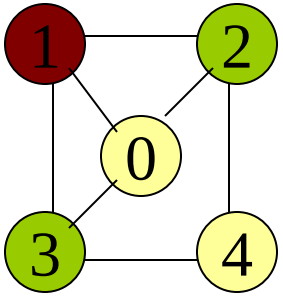
1
2
4



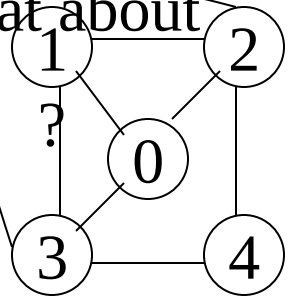
2
4



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

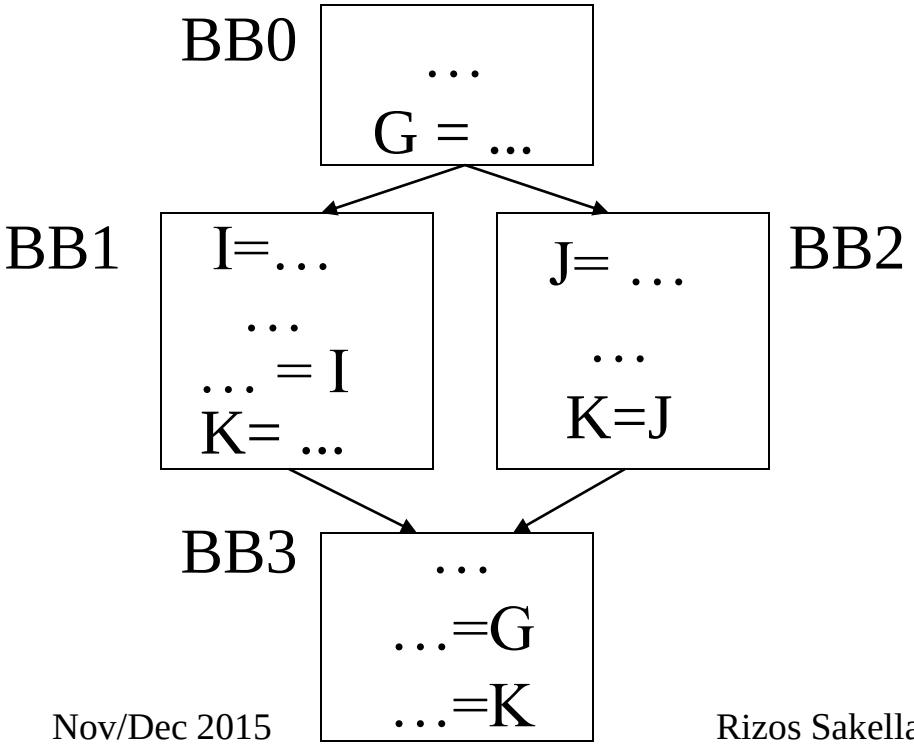
- At some point p in a basic block, a value v is live if it has been defined along a path from that basic block's ancestors to p and a path exists from p to a use of v, along which v is not redefined.

  (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

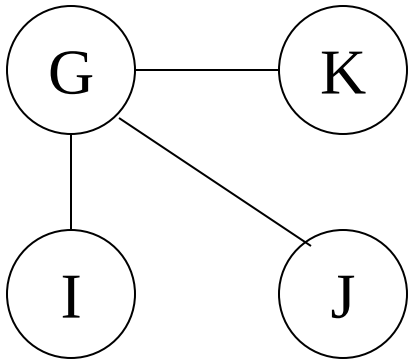
- If basic block has no successors, LIVEOUT(b) = ∅
- For all other basic blocks: LIVEOUT(b)=∪ LIVEIN(s) for all immediate successors, s, of b in the control flow graph.
- A value is in LIVEIN(b):
  - if it is used before it is defined (if it is defined) in basic block b; or
  - it is not used, nor defined,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

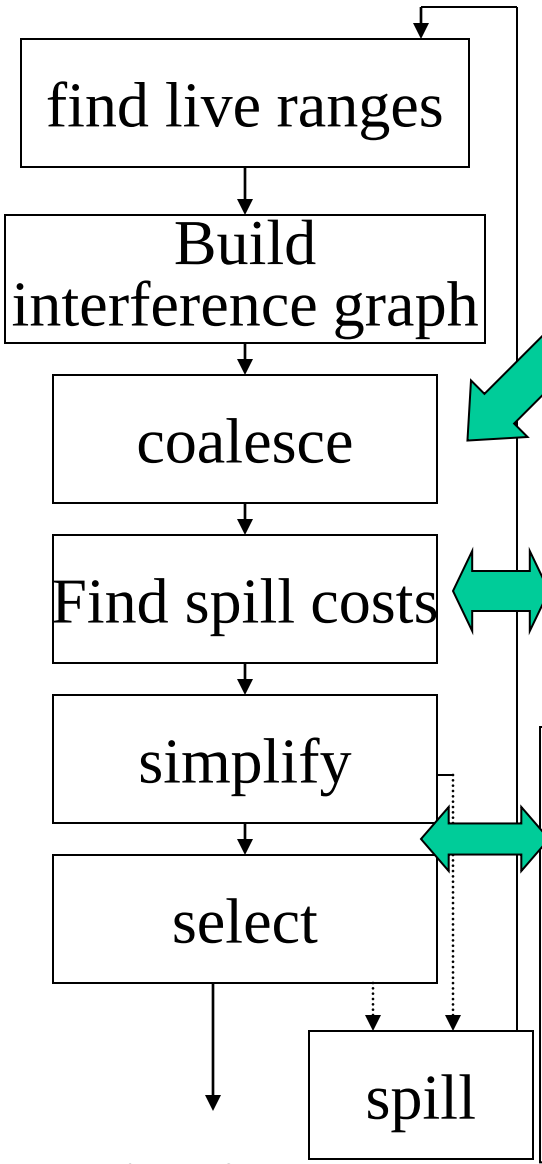
for each basic block b
    LIVENOW(b)←LIVEOUT(b)
    for each operation in b (in reverse order) of the type op lr1,lr2,lr3
        for each live_range in LIVENOW(b) except lr2 and lr3
            add an edge between live_range and lr1
        remove lr1 from LIVENOW(b)
        add lr2 and lr3 to LIVENOW(b)



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.

Conclusion (end of Part II)

- 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 part: Instruction Scheduling