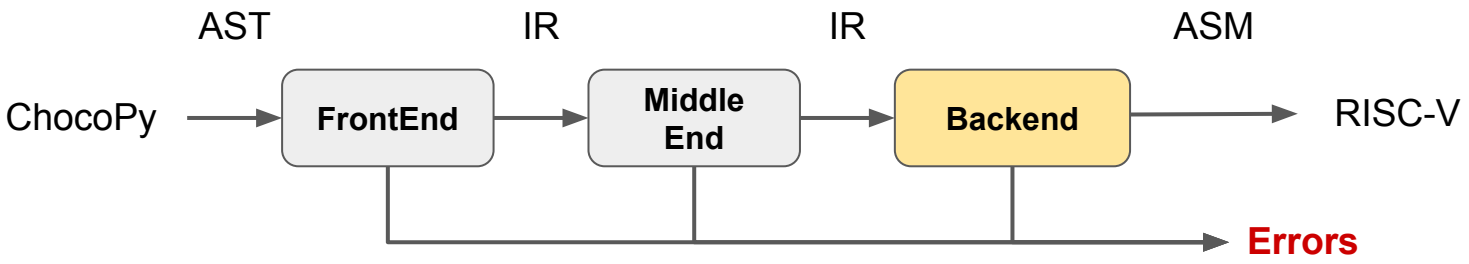
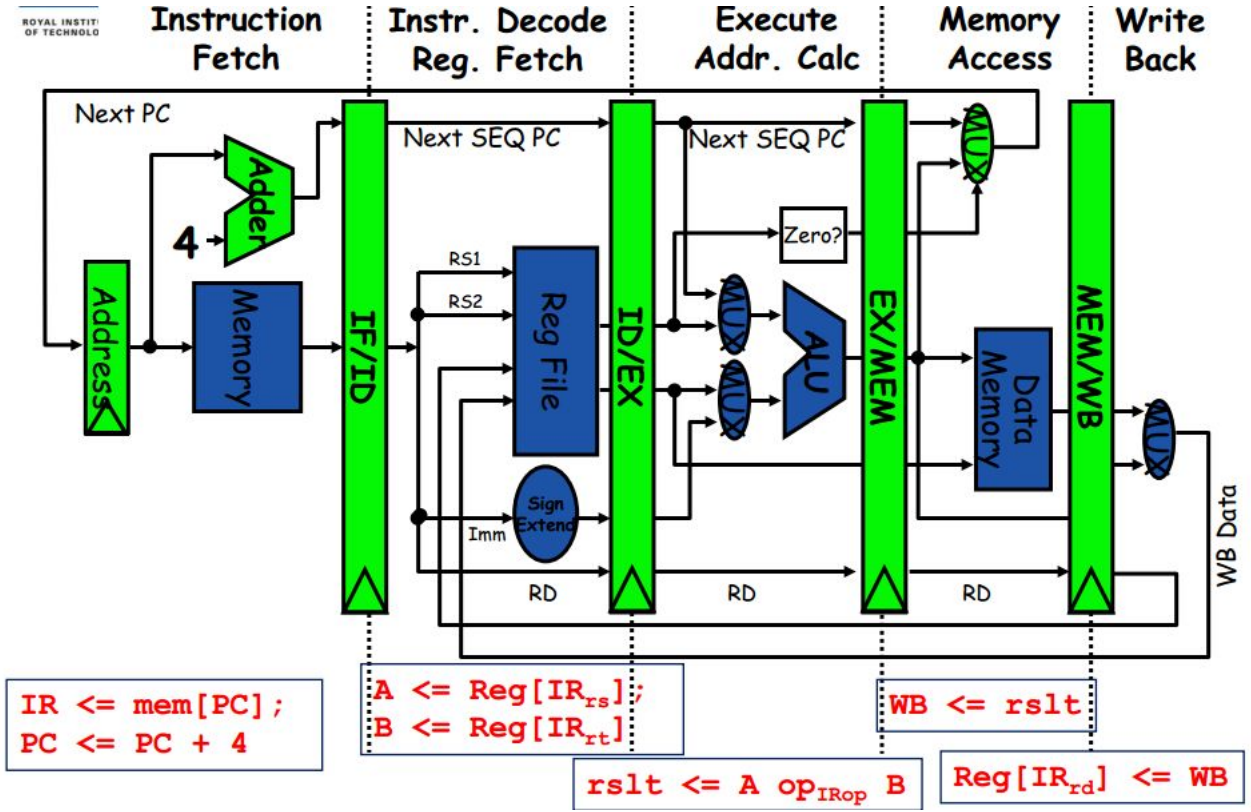
Compiling Techniques

Lecture 17: Register Allocation

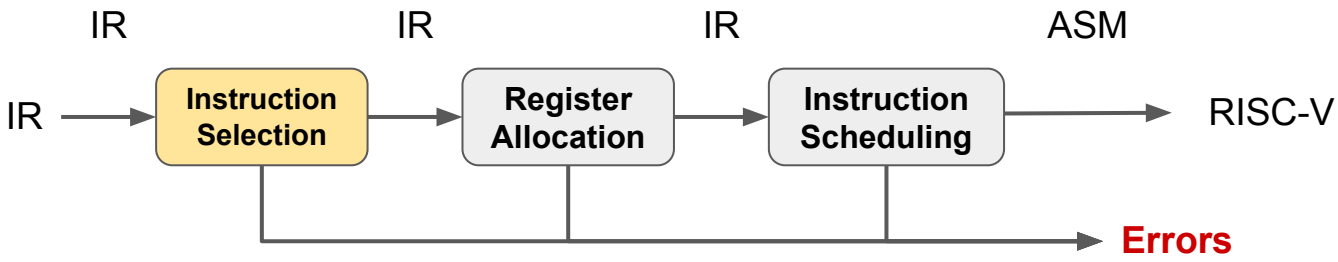
Overview


AST
IR
IR
ASM
ChocoPy
FrontEnd
Middle End
Backend
RISC-V
Errors

Review: Processor Architecture

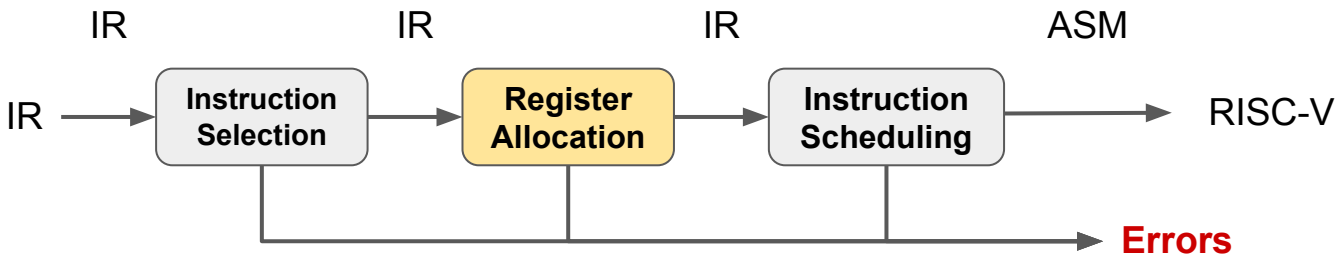


Instruction Selection



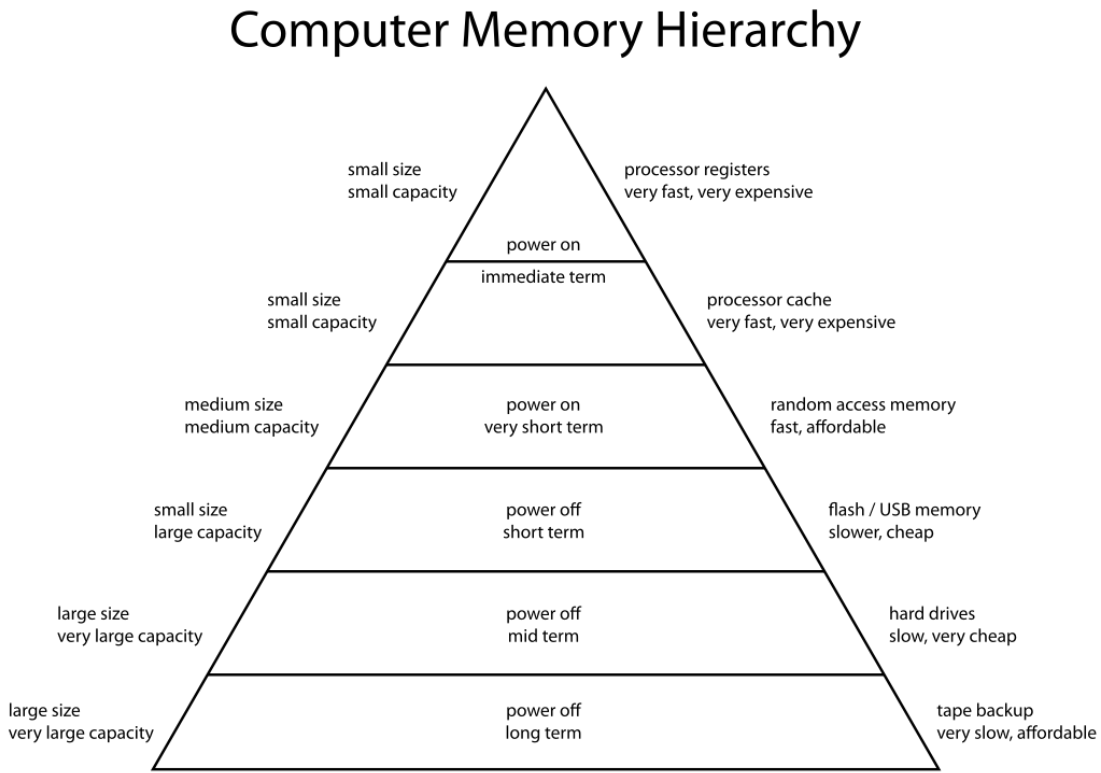
- Mapping the IR into assembly code (in our case AST to RISCV-V assembly)
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

Register Allocation

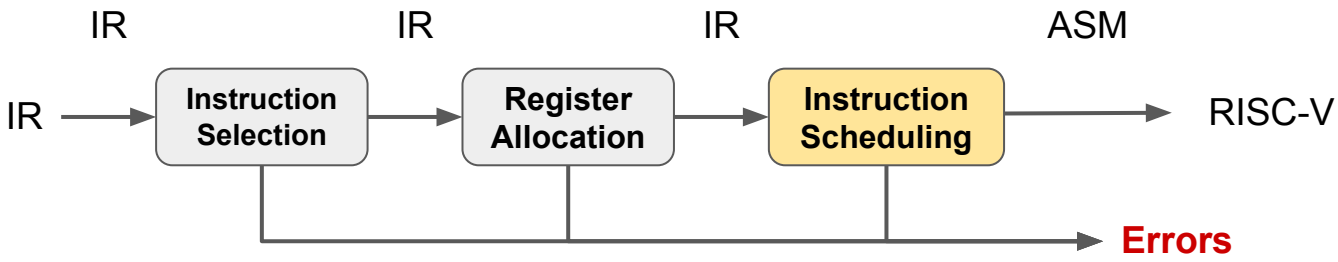


- Deciding which value reside in a register
- Minimise amount of spilling

Review: Memory Hierarchy

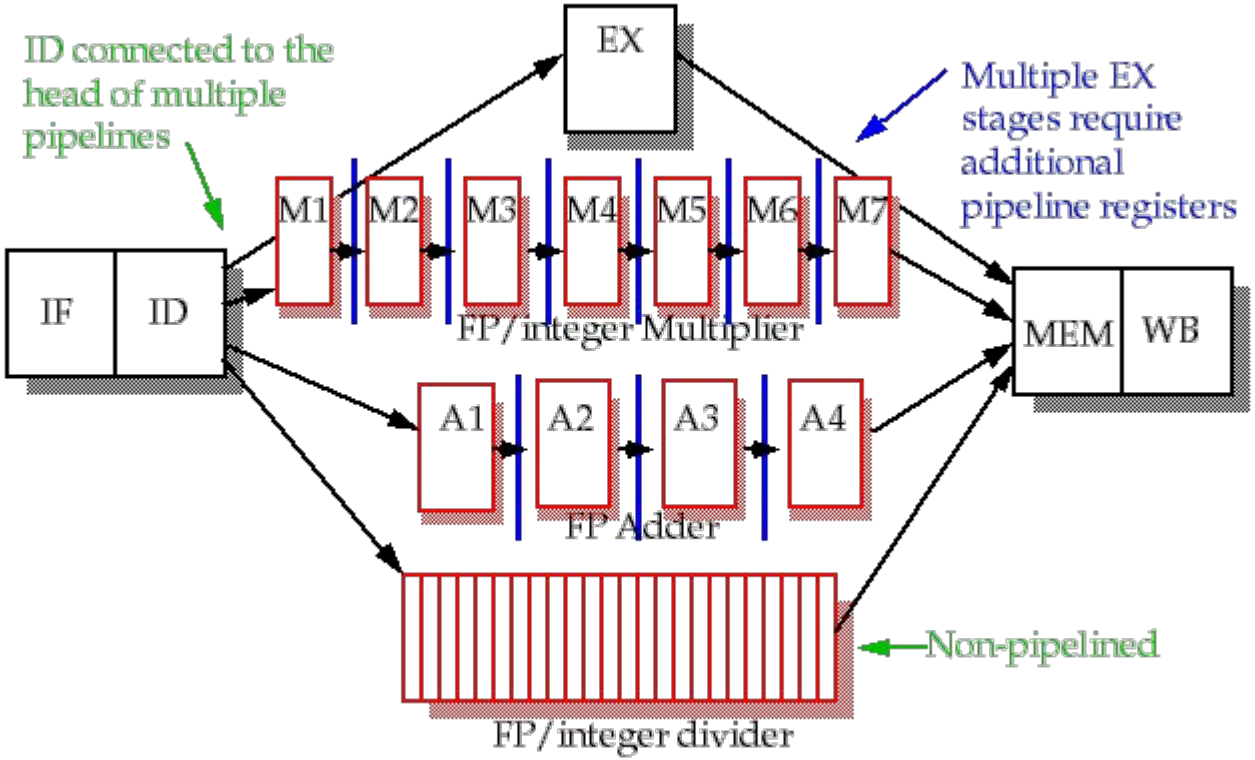


Instruction Scheduling

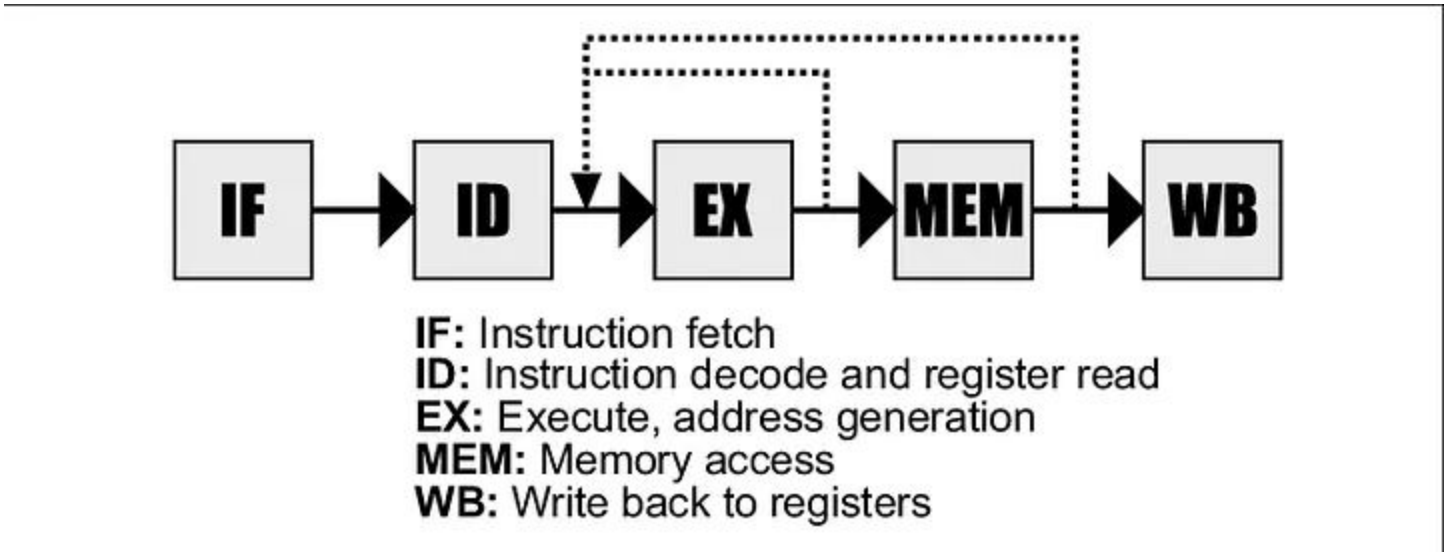


- Avoid hardware stalls and interlocks
- Reordering operations to hide latencies
- Use all functional units productively

Instruction Scheduling: Motivation



Computer Architecture: A Simple Pipeline



https://medium.com/@aylia.zulfiqar29/5-stages-of-pipeline-in-computer-architecture-dc9fca11784e

The Big Picture

These problems are tightly coupled

However, conventional wisdom says we lose little by solving these problems independently.

How hard are these problems?

- Instruction selection
  - Can make locally optimal choices, with automated tool
  - Global optimality is NP-Complete
  -
- Instruction scheduling
  - Single basic block ⇒ heuristic work quickly
  - General problem, with control flow ⇒ NP-Complete

- Register allocation
  - Single basic block, no spilling ⇒ linear time
  - Whole procedure is NP-Complete (graph colouring algorithm)

How to solve these problems?

Approximate Solutions

Will be important to define good metrics for “close”, “good”, “enough”, ...

How to solve these problems?

- Instruction selection
  - Use some form of pattern matching
  - Assume enough registers or target “**important**” values
- Instruction scheduling
  - Within a block, list scheduling is “**close**” to optimal
  - Across blocks, build framework to apply list scheduling
- Register allocation
  - Start from virtual registers & map “**enough**” into k
  - With targeting, focus on “**good**” priority heuristic

Example: Instruction Selection vs Register Allocation in Arm

- Arm has load-pair (LDP) instructions that allow you to load two registers at once:
  - ldp r1, r2, [r3, 0] — loads from address in r3 into r1 and r2
- Downside:
  - Extends live ranges of r1 or r2

Example: Instruction Scheduling vs Register Allocation

```
fdiv r1, r2, r2
…. (Long Latency Instruction)
sw r1, [r3]
lw r1, 0
add r1, r1, r1
```

Instruction Scheduling →

```
fdiv r1, r2, r2
…. (Long Latency Instruction)
lw r1, 0
add r1, r1, r1
sw r1, [r3]
```

Register Allocation

- Physical machines have limited number of registers
- Scheduling and selection typically assume infinite registers
- Register allocation and assignment from infinite to k registers

- Produce correct code that uses k (or fewer) registers
- Minimise added loads and stores
- Minimise space used to hold spilled values
- Operate efficiently:
  - $O(n)$, $O(n^2)$, but not $O(2^n)$

Register Allocation: Definitions

Allocation vs Assignment:

- *Allocation* is deciding which values to keep in registers
- *Assignment* is choosing specific registers for values

Liveness

A value is live from its definition to its last use.

Interference

Two values cannot be mapped to the same register wherever they are both live. Such values are said to **interfere**.

Live Range

The live range of a value is the set of statements at which it is live. A live range may be conservatively overestimated (e.g, just begin → end)

Register Allocation: Definitions

Spilling

Spilling saves a value from a register to memory.
That register is then free - Another value often loaded

Requires F registers to be reserved.

Clean and dirty values

A previously spilled value is **clean** if not changed since last spill.
Otherwise it is **dirty**.

A clean value can be spilled without a new store instruction.

Register Allocation Algorithms

- Static
- Linear
- Graph-Coloring Based

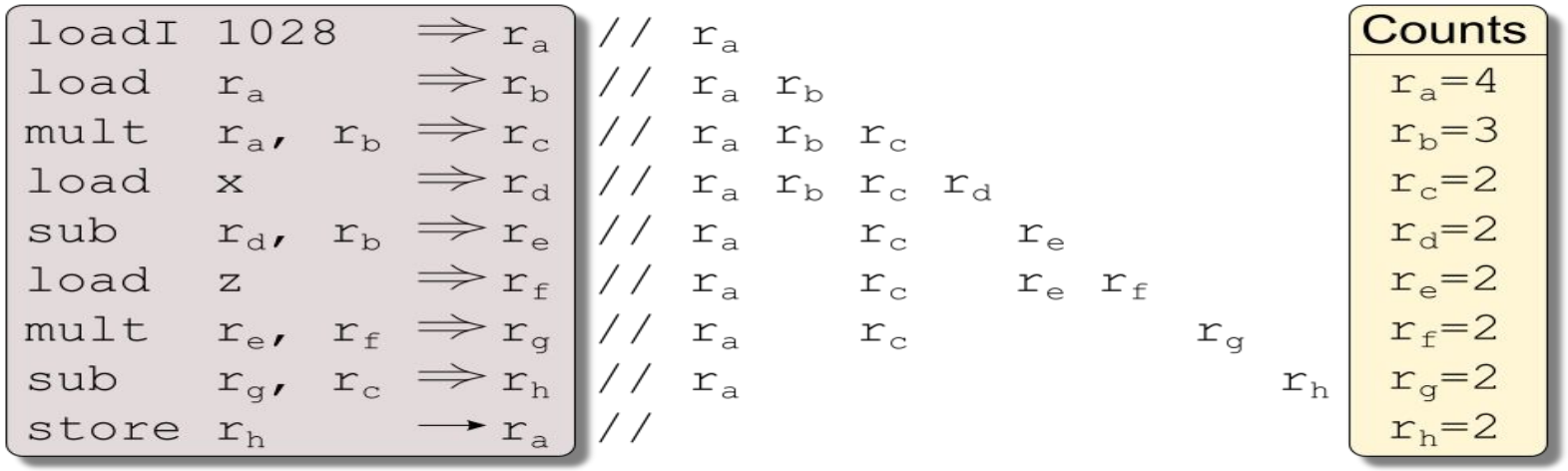
Live Ranges Reminder

```
loadI  1028    ⇒ ra   //
load   ra      ⇒ rb   //
mult   ra, rb  ⇒ rc   //
load   x       ⇒ rd   //
sub    rd, rb  ⇒ re   //
load   z       ⇒ rf   //
mult   re, rf  ⇒ rg   //
sub    rg, rc  ⇒ rh   //
store  rh      → ra   //
```

Top-Down Register Allocation

- Idea: Put the most frequently used variables in register
- Save some backup registers
- Whenever we use a variable that didn’t get a register, use the backup registers to load in, and then store out the variable

Top-Down Register Allocation



Linear-Scan Register Allocation

- Idea: We can re-use registers, with a greedy approach
- Assign variables to registers in one pass through the variables

Linear-Scan Register Allocation

Register 1	Register 2	Register 3

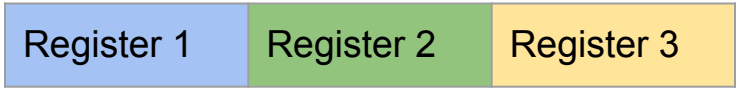
a							
a	b						
a	b	c					
a	b	c	d				
a		c		e			
a		c		e	f		
a		c				g	
a							h

Linear-Scan Register Allocation

Register 1	Register 2	Register 3

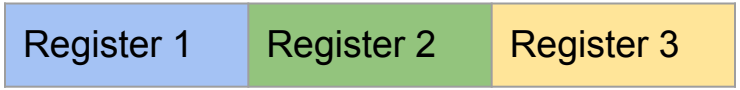
a							
a	b						
a	b	c					
a	b	c	d				
a		c		e			
a		c		e	f		
a		c				g	
a							h

Linear-Scan Register Allocation



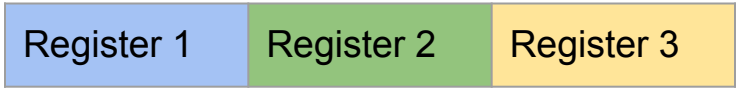
a							
a	b						
a	b	c					
a	b	c	d				
a		c		e			
a		c		e	f		
a		c				g	
a							h

Linear-Scan Register Allocation



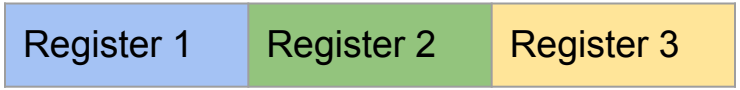
a							
a	b						
a	b	c					
a	b	c	d				
a		c		e			
a		c		e	f		
a		c				g	
a							h

Linear-Scan Register Allocation



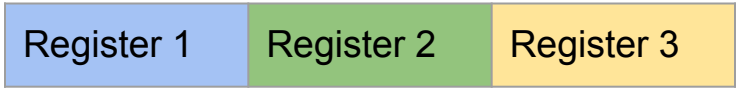
a							
a	b						
a	b	c					
a	b	c	d (need to spill!)				
a		c		e			
a		c		e	f		
a		c				g	
a							h

Linear-Scan Register Allocation



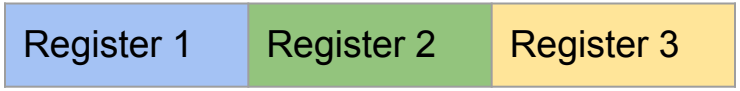
a							
a	b						
a	b	c					
a	b	c	d (need to spill!)				
a		c		e			
a		c		e	f		
a		c				g	
a							h

Linear-Scan Register Allocation



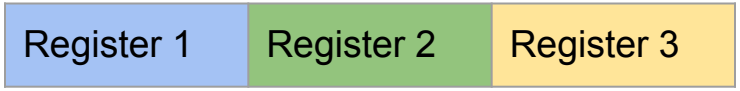
a							
a	b						
a	b	c					
a	b	c	d (need to spill!)				
a		c		e			
a		c		e	f (need to spill)		
a		c				g	
a							h

Linear-Scan Register Allocation



a							
a	b						
a	b	c					
a	b	c	d (need to spill!)				
a		c		e			
a		c		e	f (need to spill)		
a		c				g	
a							h

Linear-Scan Register Allocation



a							
a	b						
a	b	c					
a	b	c	d (need to spill!)				
a		c		e			
a		c		e	f (need to spill)		
a		c				g	
a							h

More Optimal Register Allocation

Local allocation does not capture reuse of values across multiple blocks

Most modern, global allocators use a graph-colouring paradigm

- Build a "**conflict graph**" or "**interference graph**"
  - Data flow based liveness analysis for interference
- Find a **k-colouring** for the graph, or change the code to a nearby problem that it can k-colour
- NP-complete under nearly all assumptions[1]

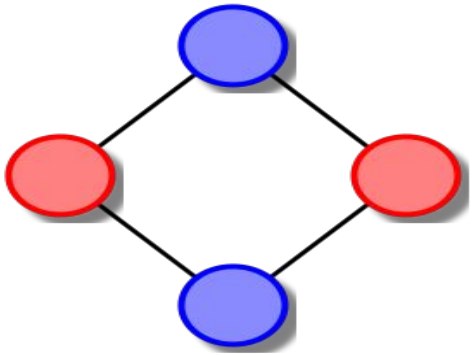
[1] Local allocation is NP-complete with dirty vs clean

Global register allocation: algorithm sketch

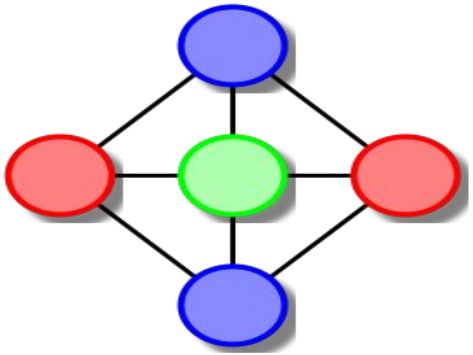
- From live ranges construct an interference graph
- Colour interference graph so that no two neighbouring nodes have same colour
- If graph needs more than k colours - transform code
  - Coalesce merge-able copies
  - Split live ranges
  - Spill
- Colouring is NP-complete so we will need heuristics
- Map colours onto physical registers

Global register allocation: Graph Coloring

A graph G is said to be **k-colourable** if the nodes can be labeled with integers 1 ... k so that no edge in G connects two nodes with the same label



2-colourable



3-colourable

Interference Graph

The interference graph, $G = (N, E)$

- Nodes in G represent values, or live ranges
- Edges in G represent individual interferences
- $\forall x, y \in N, x \rightarrow y \in E$ *iff* x and y interfere

A *k-colouring* of G can be mapped into an allocation to k registers

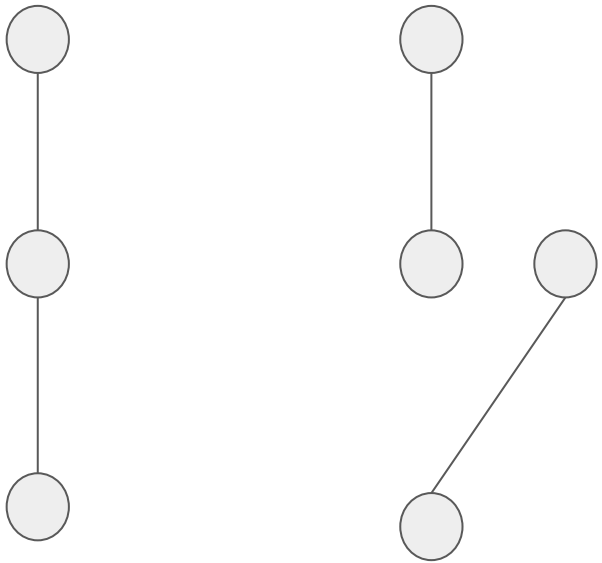
Two values interfere wherever they are both live
Two live ranges interfere if their values interfere at any point

Chaitin’s Algorithm

1. While ∃ vertices with < k neighbours in G
   - Pick any vertex n such that n° < k and put it on the stack
   - Remove n and all edges incident to it from G
2. If G is non-empty (n° >= k, ∃ n ∋ G) then:
   - Pick vertex n (heuristic), spill live range of n
   - Remove vertex n and edges from GI, put n on “spill list”
   - Goto step 1
3. If the spill list is not empty, insert spill code, then rebuild the interference graph and try to allocate, again
4. Otherwise, successively pop vertices of the stack and colour them in the lowest colour not used by some neighbour

Insert Spill Code

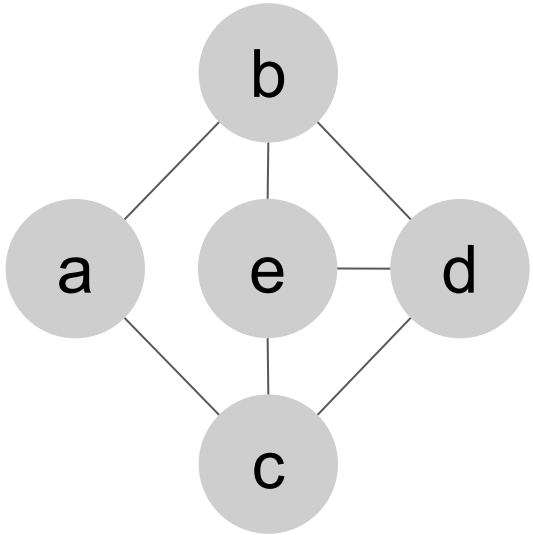
- Create two nodes in the graph

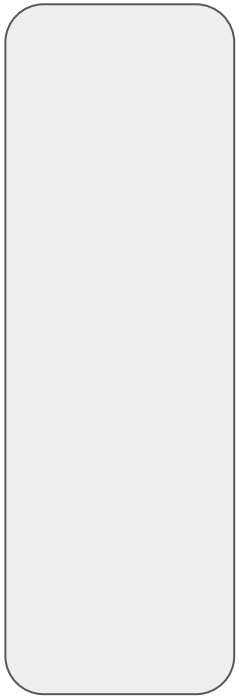


In-Person Demo!

1) Take some strings, 1 pen, 1 sheet of paper
2) Form a conflict graph using the strings
3) Do Chaitin's Algorithm

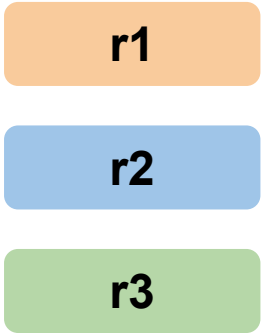
Global Register Allocation: Chaitin's Algorithm





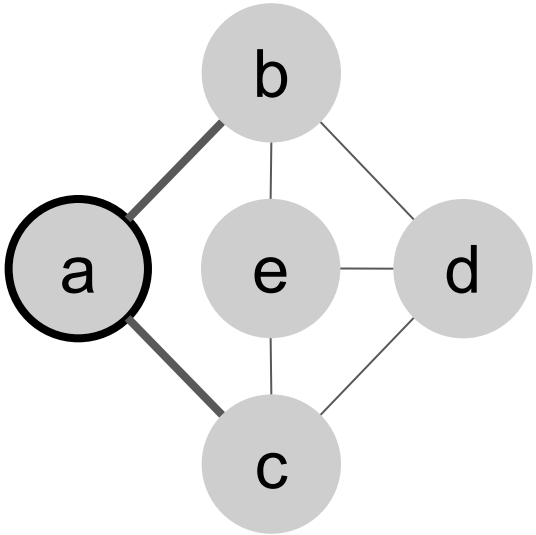
Stack

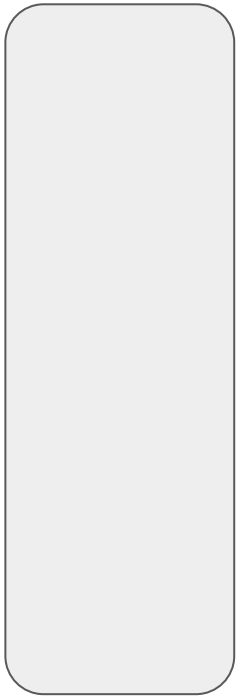
Colour with k = 3 colours



Colours

Global Register Allocation: Chaitin's Algorithm





Stack

a° = 2 < k Choose a

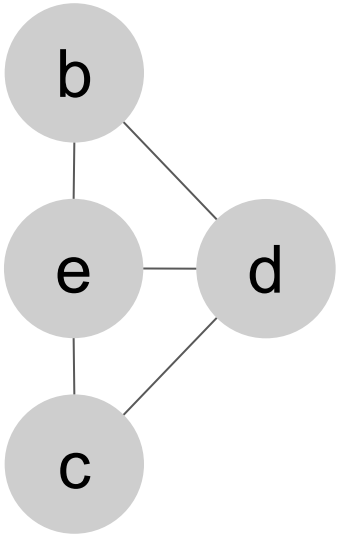
r1

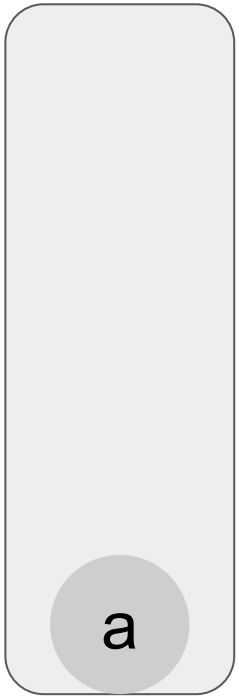
r2

r3

Colours

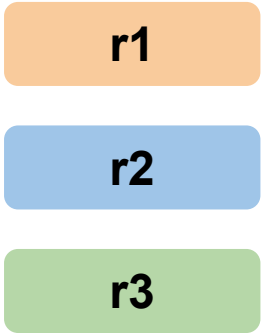
Global Register Allocation: Chaitin's Algorithm





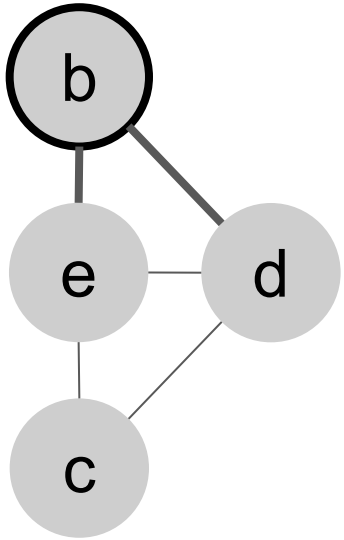
Stack

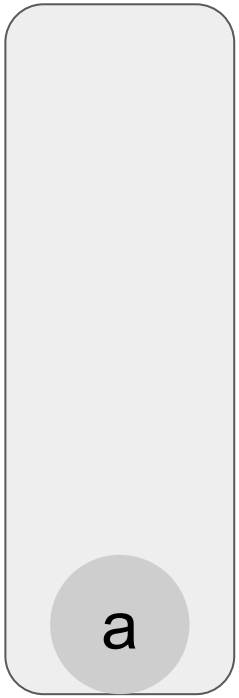
Push a and remove from graph



Colours

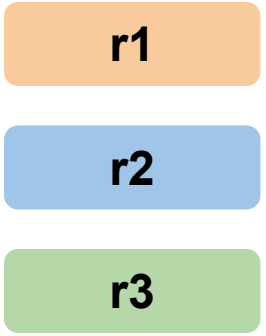
Global Register Allocation: Chaitin's Algorithm





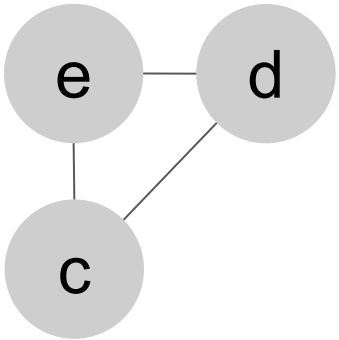
Stack

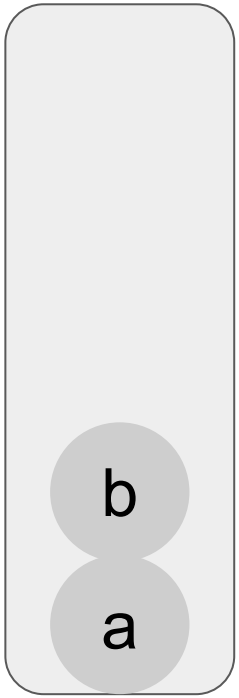
$b^\circ = 2 < k$ and
$c^\circ = 2 < k$
Choose b



Colours

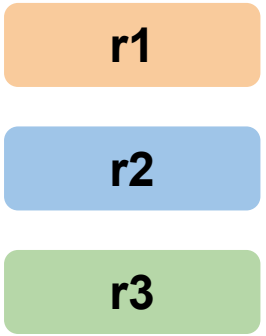
Global Register Allocation: Chaitin's Algorithm





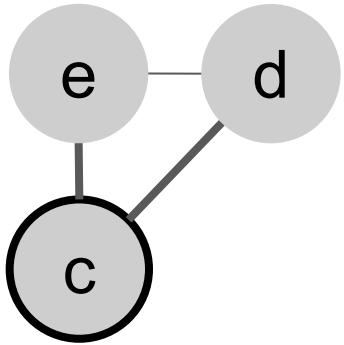
Stack

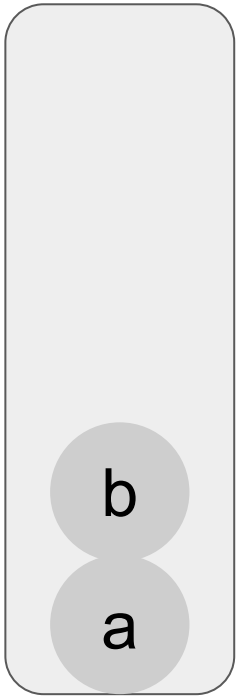
Push b and remove from graph



Colours

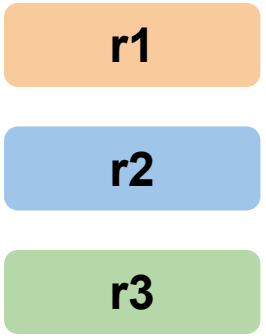
Global Register Allocation: Chaitin's Algorithm





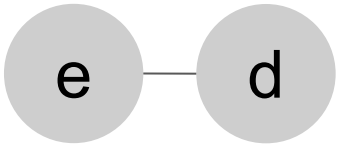
Stack

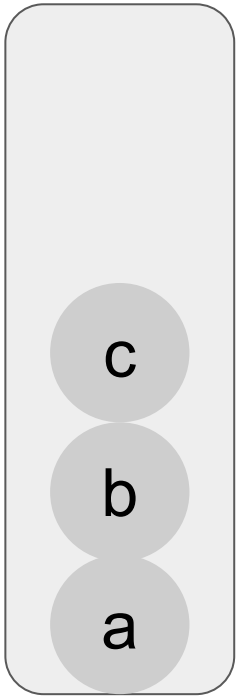
c° = 2 < k,
d° = 2 < k, and
e° = 2 < k
Choose c



Colours

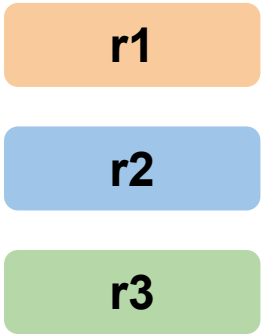
Global Register Allocation: Chaitin's Algorithm





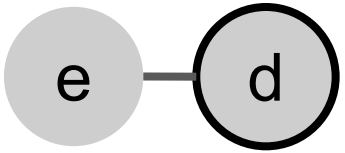
Stack

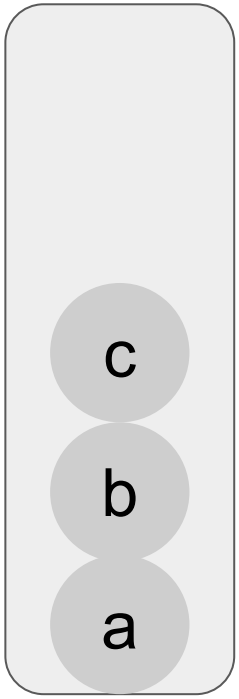
Push c and remove from graph



Colours

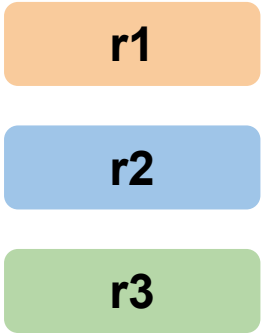
Global Register Allocation: Chaitin's Algorithm





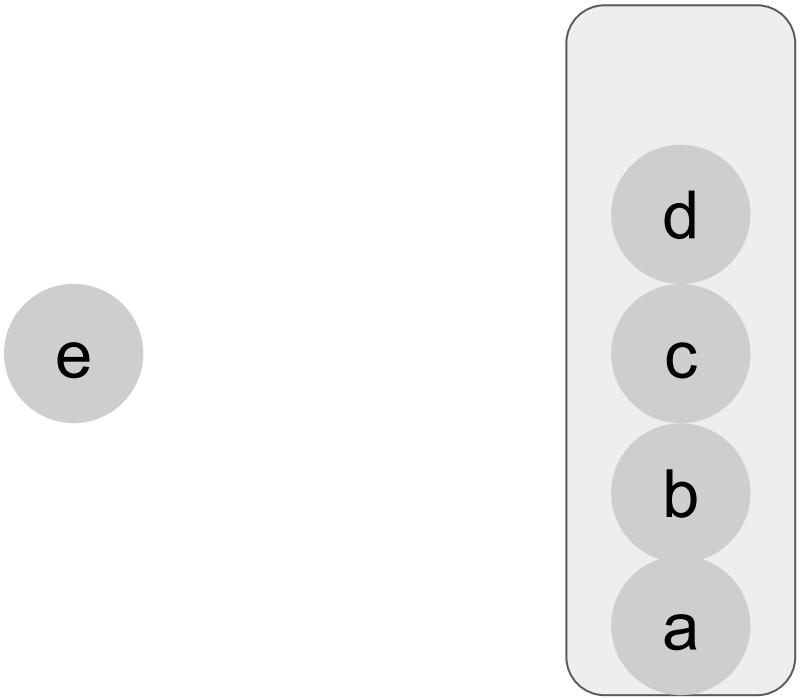
Stack

d° = 1 < k and
e° = 1 < k
Choose d

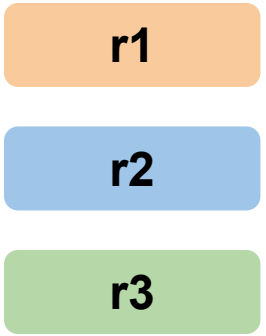


Colours

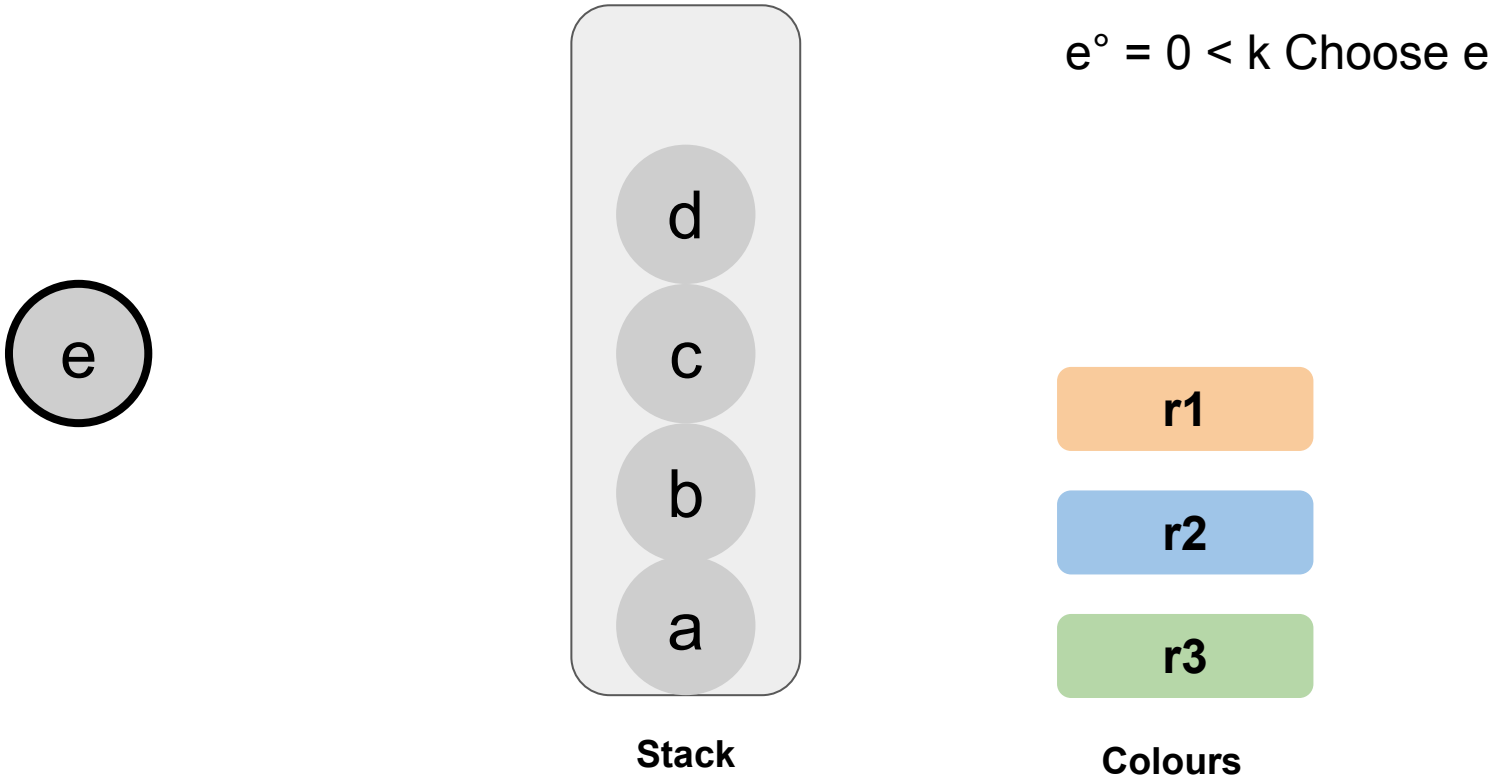
Global Register Allocation: Chaitin's Algorithm



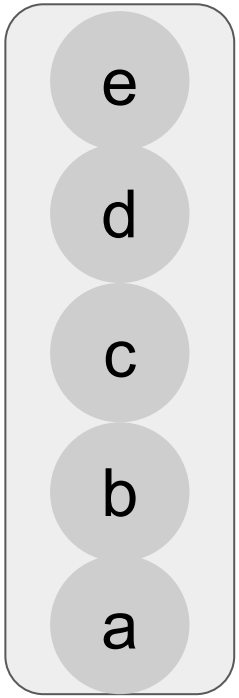
Push d and remove from graph



Global Register Allocation: Chaitin's Algorithm

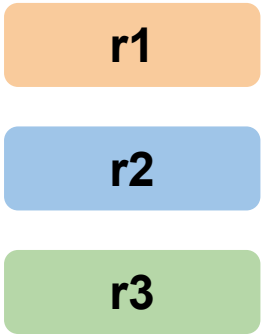


Global Register Allocation: Chaitin's Algorithm



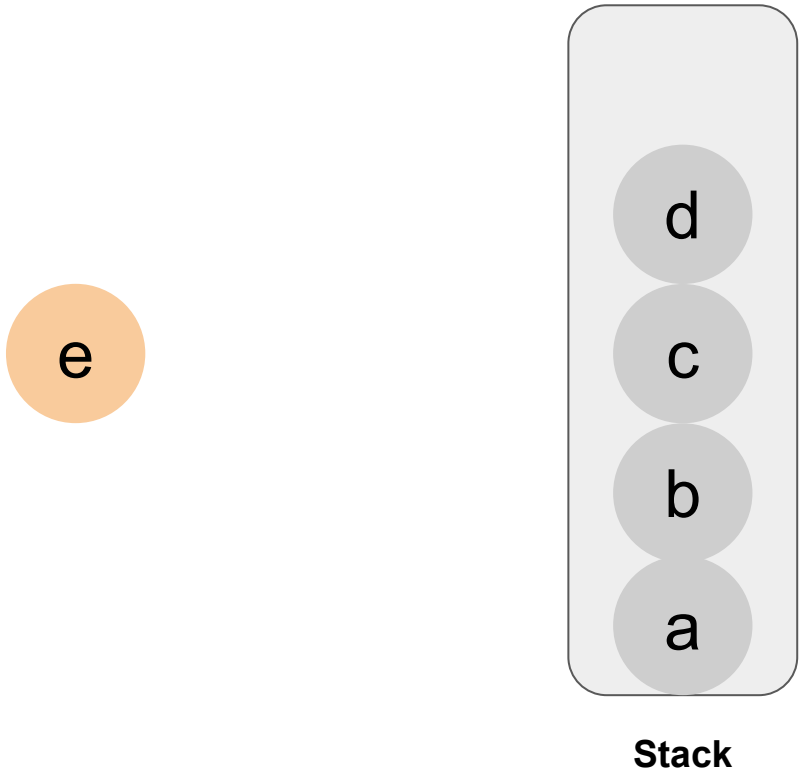
Stack

Push e and remove from graph

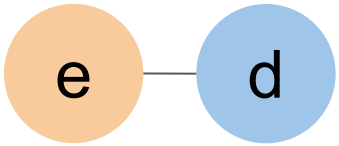


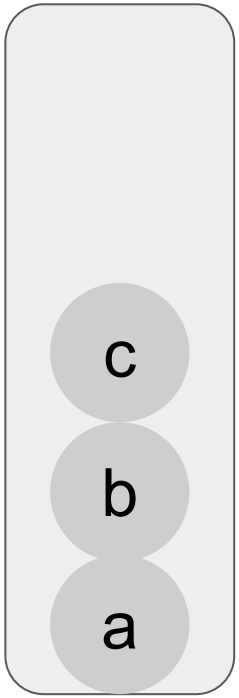
Colours

Global Register Allocation: Chaitin's Algorithm



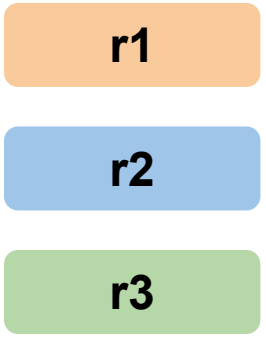
Global Register Allocation: Chaitin's Algorithm





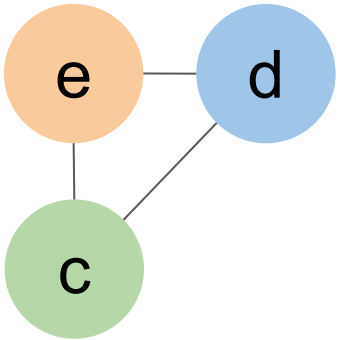
Stack

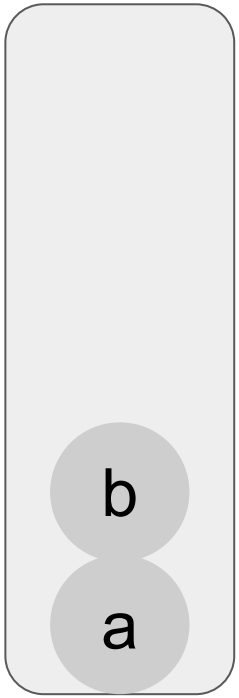
Pop d, neighbours use red, choose blue



Colours

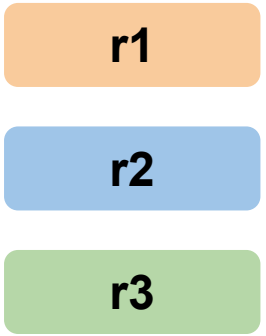
Global Register Allocation: Chaitin's Algorithm





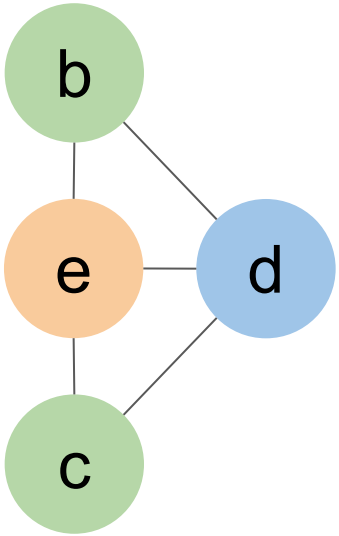
Stack

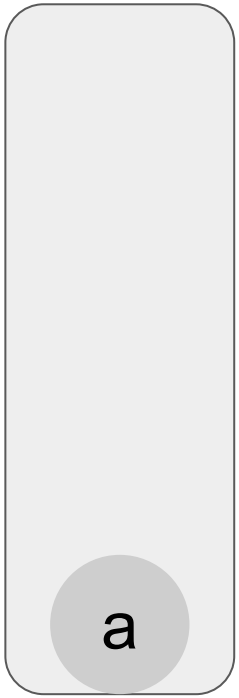
Pop c, neighbours use red and blue choose green



Colours

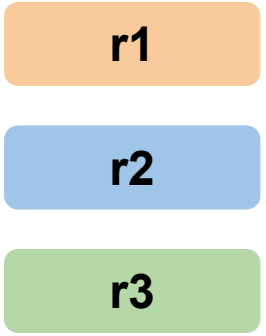
Global Register Allocation: Chaitin's Algorithm





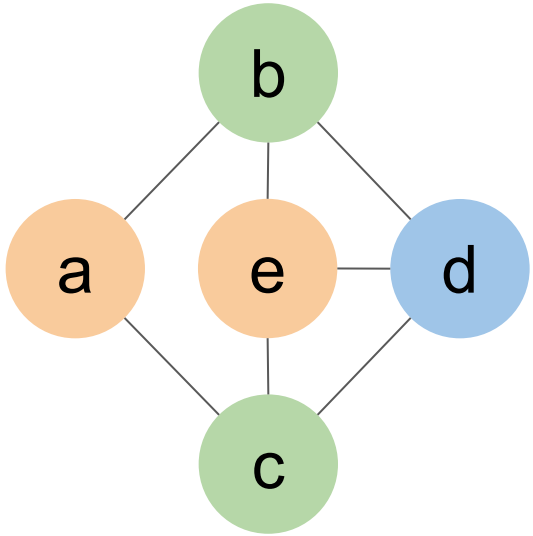
Stack

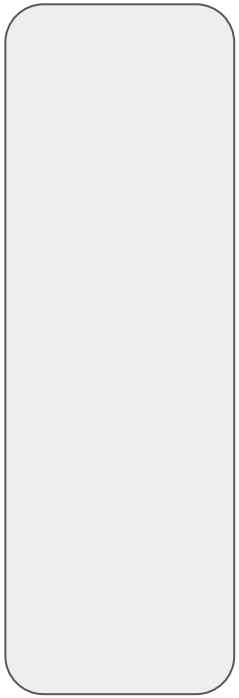
Pop c, neighbours use red and blue choose green



Colours

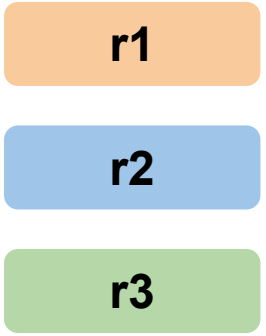
Global Register Allocation: Chaitin's Algorithm





Stack

Pop a, neighbours use blue choose red

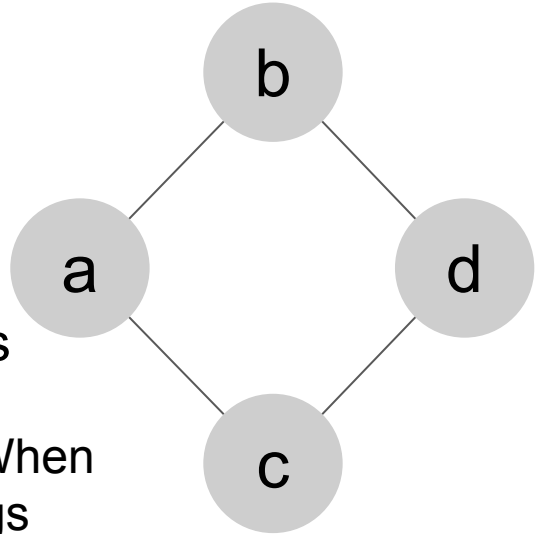


Colours

Global Register Allocation: Optimistic Colouring

If Chaitin's algorithm reaches a state where every node has k or more neighbours, it chooses a node to spill.

Example of Chaitin overzealous spilling
k = 2
Graph is 2-colourable
Chaitin must immediately spill one of these nodes



Briggs said, take that same node and push it on the stack! When you pop it off, a colour might be available for it! Chaitin-Briggs algorithm uses this to colour that graph

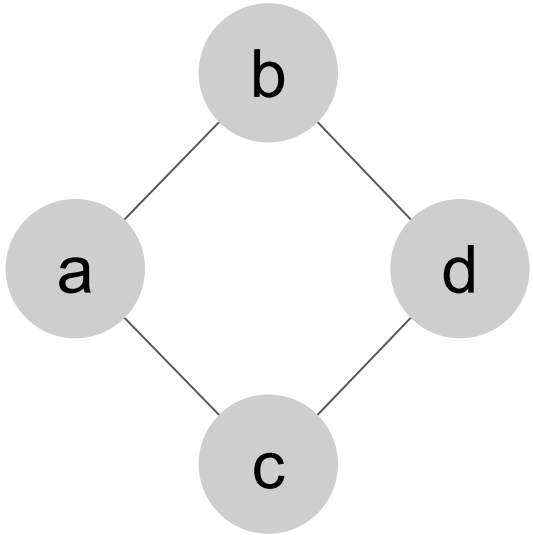
Global register allocation: Chaitin-Briggs algorithm

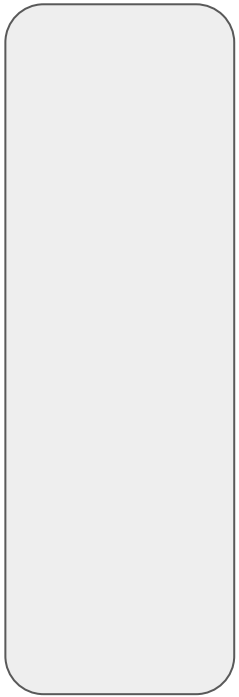
- While $\exists$ vertices with < k neighbours in G
  - Pick any vertex n such that n° < k and put it on the stack
  - Remove n and all edges incident to it from GI
- If G is non-empty (n° >= k, $\forall n \in G$) then:
  - Pick vertex n (heuristic) (Do not spill)
  - Remove vertex n from GI, put n on stack (Not spill list)
  - Goto step 1
- Otherwise, successively pop vertices off the stack and colour them in the lowest colour not used by some neighbour
  - If some vertex cannot be coloured, then pick an uncoloured vertex to spill, spill it, and restart at step 1

Distributed Demo!

1) Form groups of ~5
2) Take some strings, 1 pen, 1 sheet of paper
3) Form a conflict graph using the strings
4) Do Chaitin’s Algorithm (for 2 registers)

Global Register Allocation: Chaitin-Briggs Algorithm





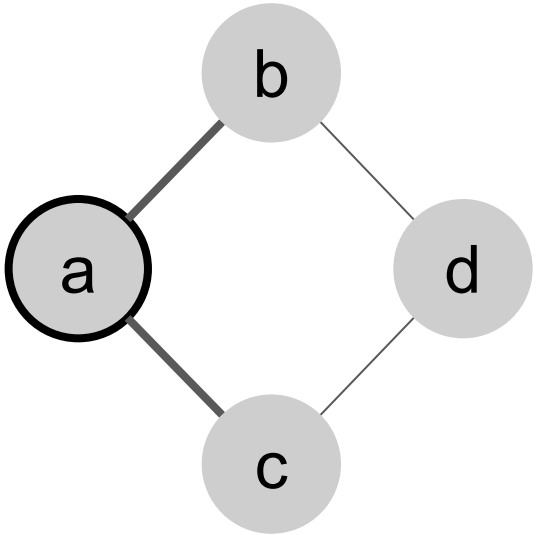
Stack

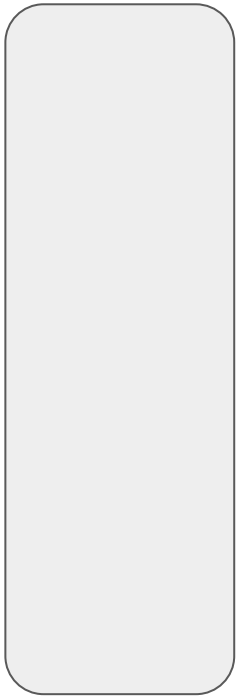
r1

r2

Colours

Global Register Allocation: Chaitin-Briggs Algorithm

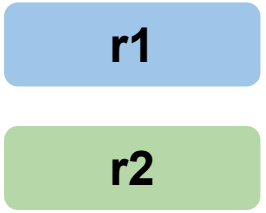




Stack

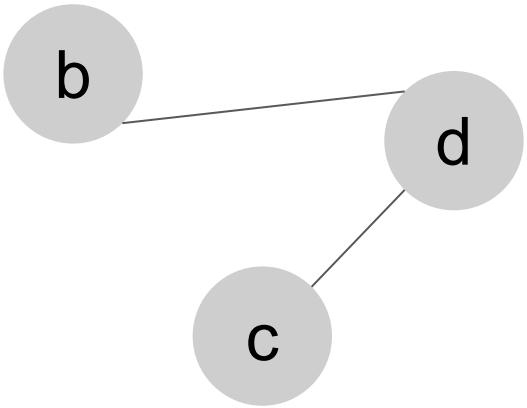
$a^\circ = 2 \geq k$

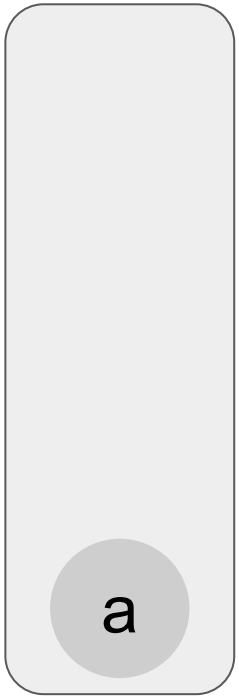
Don't Spill, Choose a!



Colours

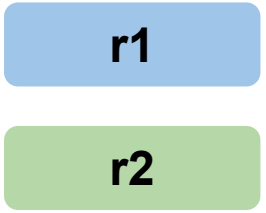
Global Register Allocation: Chaitin-Briggs Algorithm





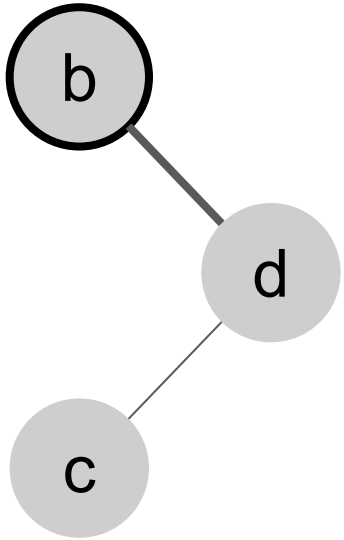
Stack

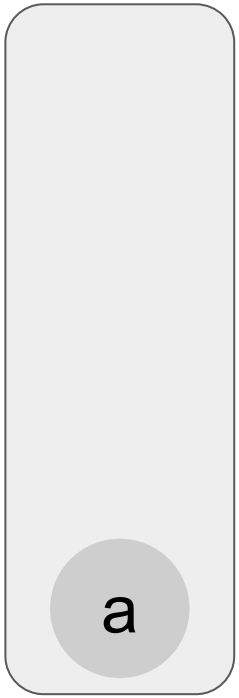
Push a and remove the graph!



Colours

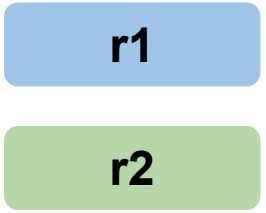
Global Register Allocation: Chaitin-Briggs Algorithm





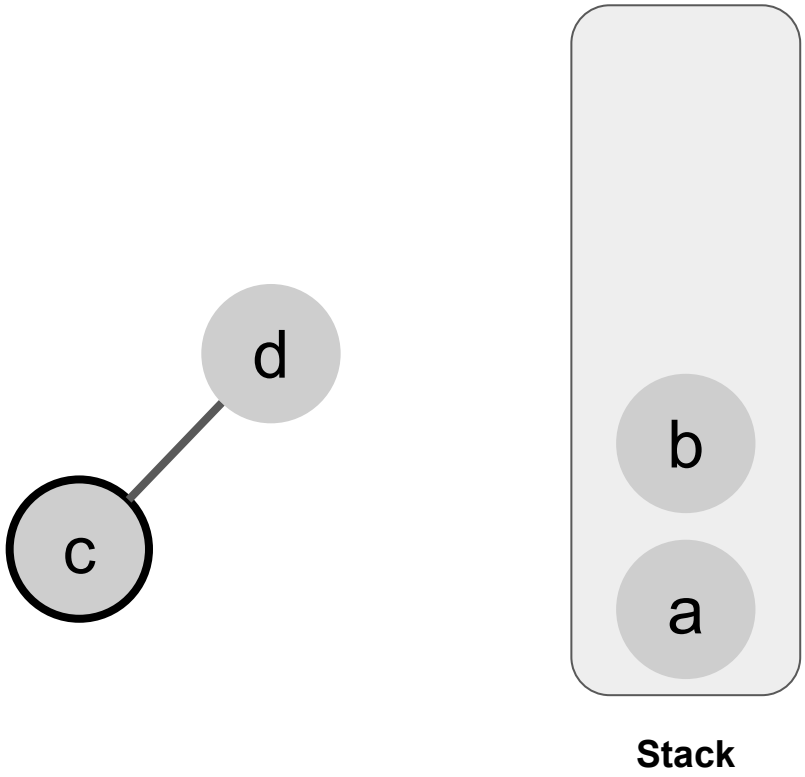
Stack

$b^\circ = 1 < k$ and
$c^\circ = 1 < k$
Choose b

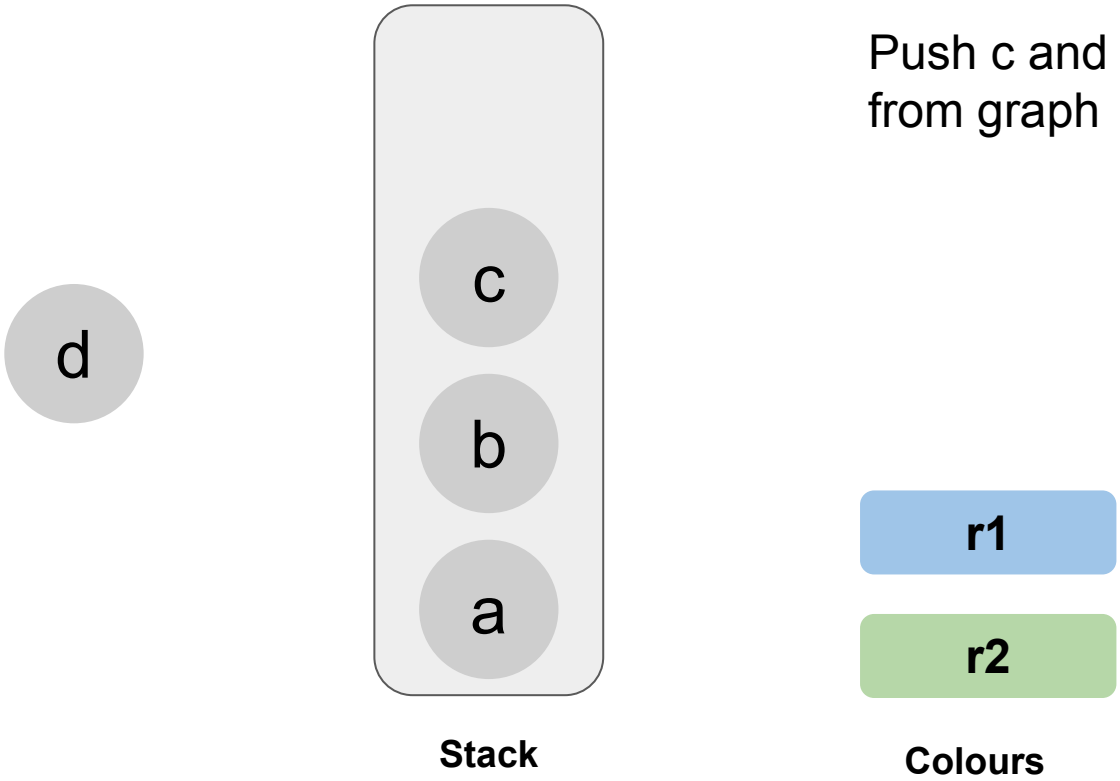


Colours

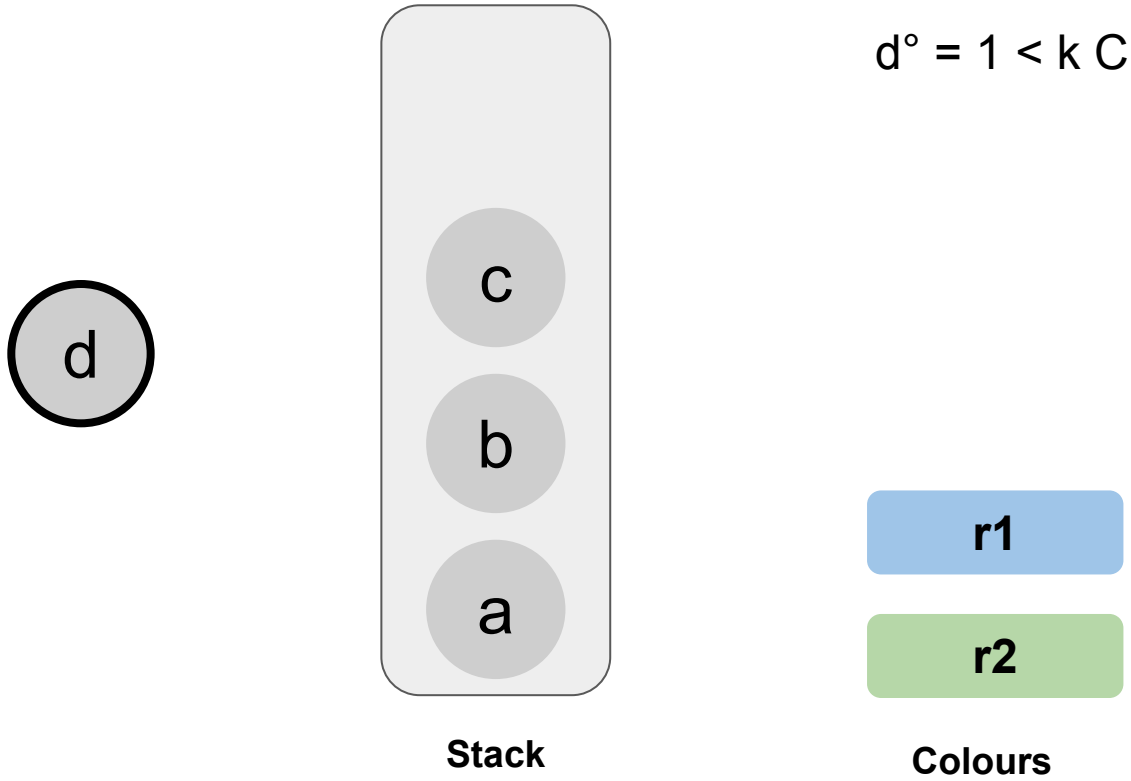
Global Register Allocation: Chaitin-Briggs Algorithm



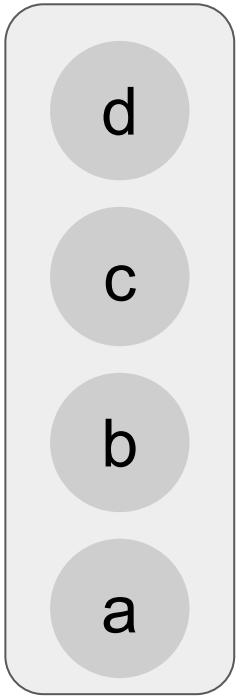
Global Register Allocation: Chaitin-Briggs Algorithm



Global Register Allocation: Chaitin-Briggs Algorithm

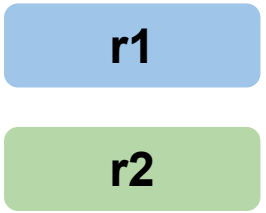


Global Register Allocation: Chaitin-Briggs Algorithm



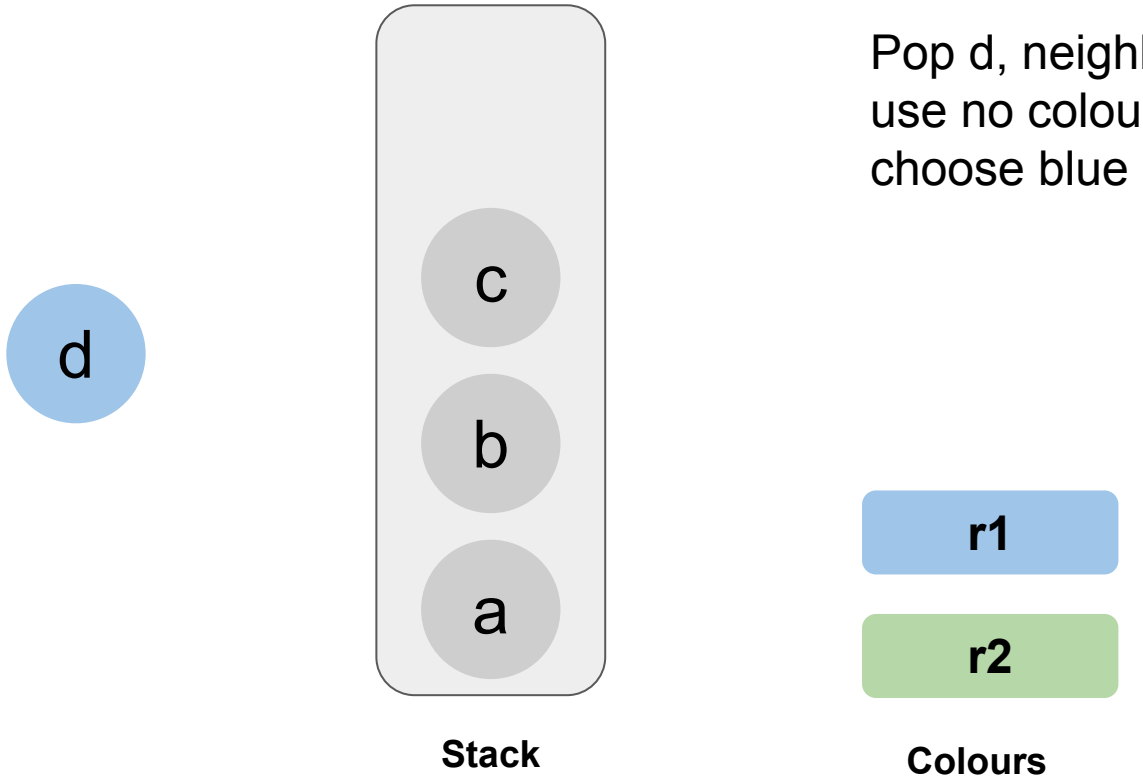
Stack

Push d and remove from graph

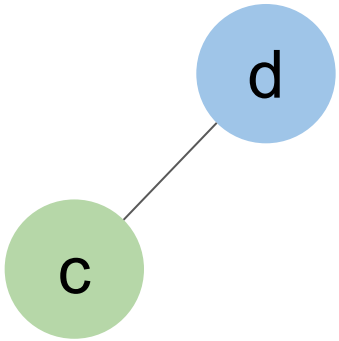


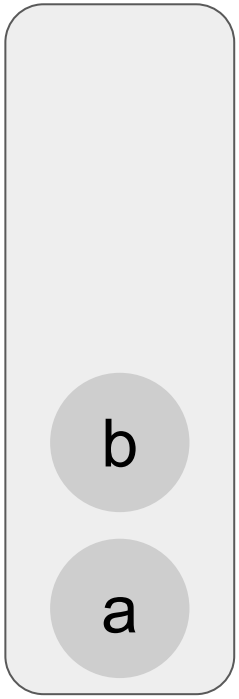
Colours

Global Register Allocation: Chaitin-Briggs Algorithm



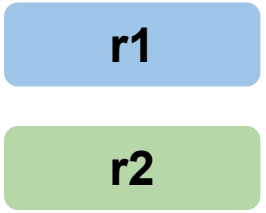
Global Register Allocation: Chaitin-Briggs Algorithm





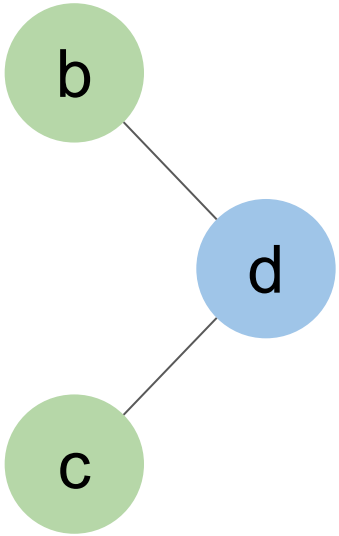
Stack

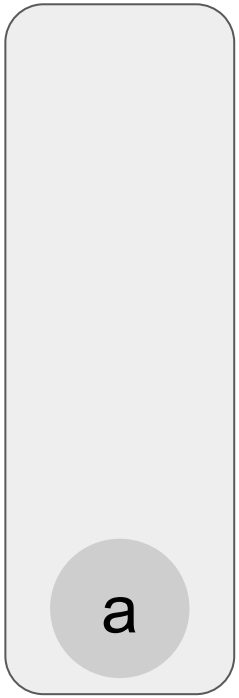
Pop c, neighbours use blue choose green



Colours

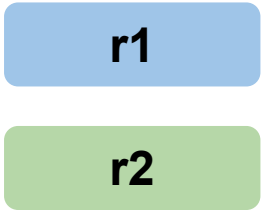
Global Register Allocation: Chaitin-Briggs Algorithm





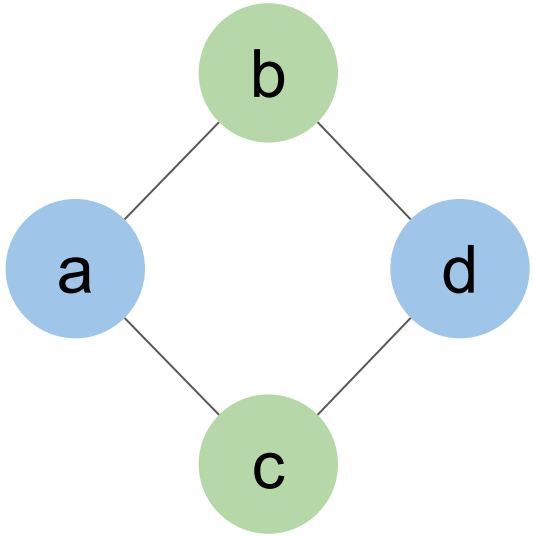
Stack

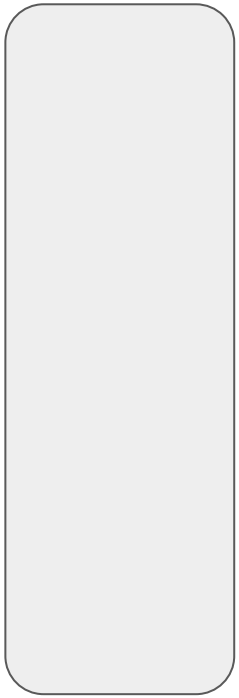
Pop b, neighbours use blue choose green



Colours

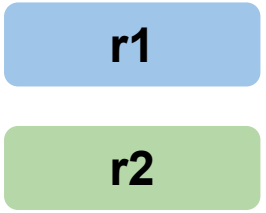
Global Register Allocation: Chaitin-Briggs Algorithm





Stack

Pop a, neighbours use green choose blue



Colours

Global register allocation: Spill Candidates

- Minimise spill cost/degree
- Spill cost is the loads and stores needed. Weighted by scope - i.e. avoid inner loops
- The higher the degree of a node to spill the greater the chance that it will help colouring
- Negative spill cost load and store to same memory location with no other uses
- Infinite cost - definition immediately followed by use. Spilling does not decrease live range

Register Allocation — Real World Concerns

- Caller/Callee save registers
- Sub-register parts (e.g., byte access)
  - E.g., x86, registers have different names
- Registers with a particular purpose (e.g., stack, function arguments)
- Instructions with fixed register arguments (e.g., branch and link)
- Re-materialisation - if easy to recreate a value do so rather than spill

Managing Live Ranges

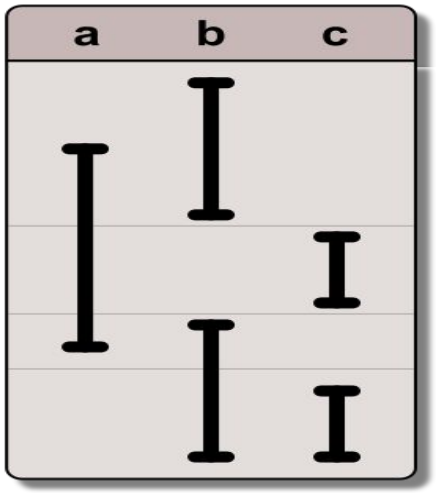
- Splitting live ranges
- Coalesce

Managing Live Ranges: Live Range Splitting

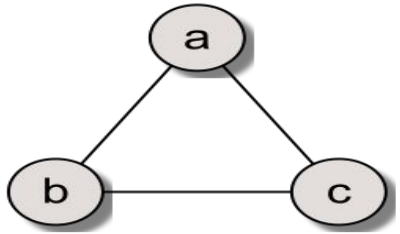
- A whole live range may have many interferences, but perhaps not all at the same time
- Split live range into two variables connected by copy
- Can reduce degree of interference graph
- Smart splitting allows spilling to occur in “cheap” regions
- **Intuition**: Allow a variable to ‘move’ between registers during its lifetime

Managing Live Ranges

Splitting example: Non contiguous live ranges - cannot be 2 coloured



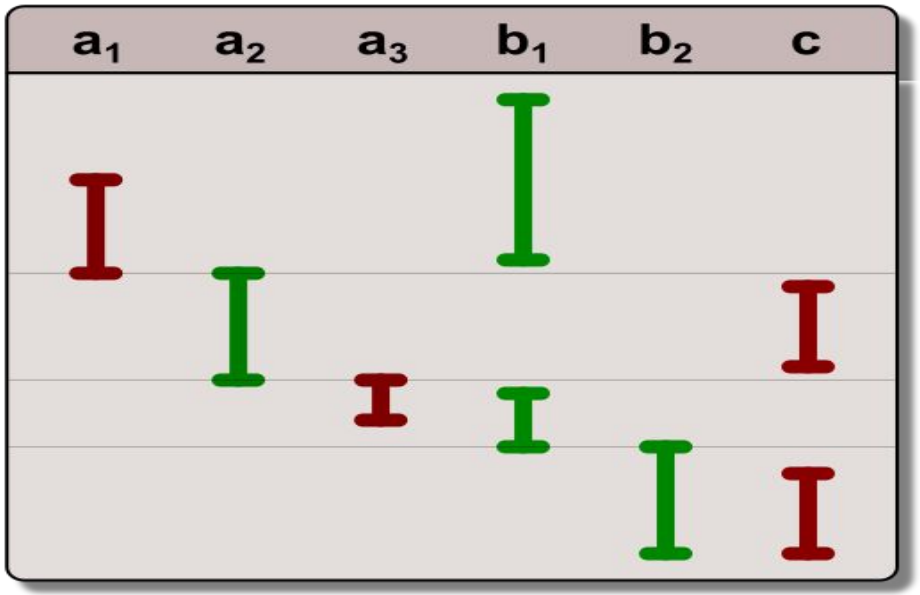
Live ranges



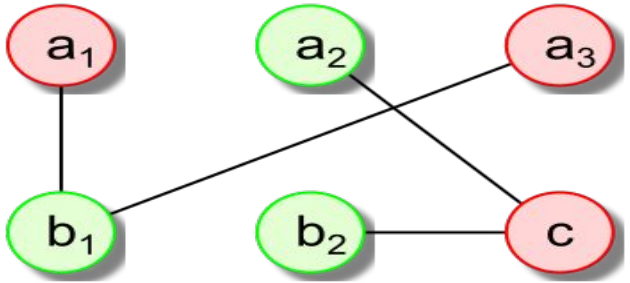
Interference Graph

Managing Live Ranges: Live Range Splitting

Splitting example: Non contiguous live ranges - can be 2 coloured



Live ranges



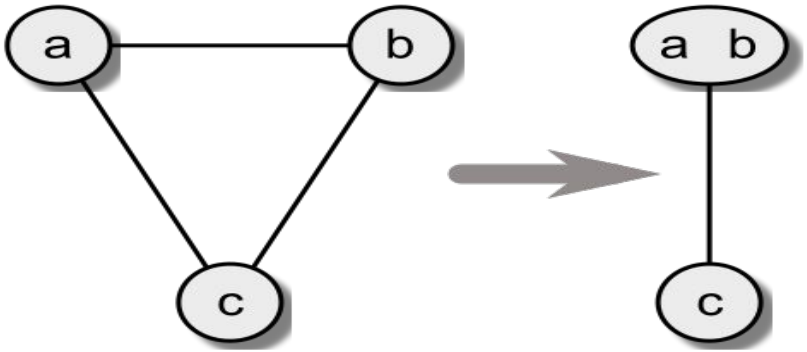
Interference Graph

Managing Live Ranges: Coalescing

If two ranges don't interfere and are connected by a copy coalesce into one – opposite of splitting. Reduces degree of nodes that interfered with both

If $x := y$ *and* $x \rightarrow y \in G_I$ *then can combine* LR_x *and* LR_y

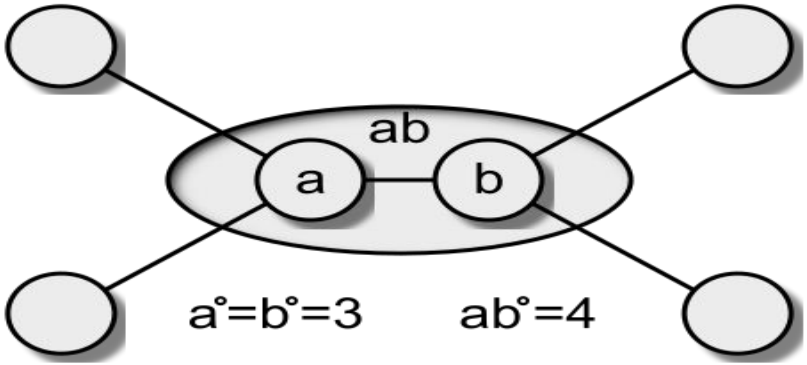
- Eliminates the copy operation
- Reduces degree of LRs that interfere with both *x* and *y*
- If a node interfered with both before, coalescing helps because it reduces degree, often applied before colouring takes place



Managing Live Ranges: Coalescing

Coalescing can make the graph harder to color

- Typically, LR_{xy} ° > *max* (LR_x °, LR_y °)
- If *max* (*LRx* °, *LRy* °) < *k* and *k* < LR_{xy} ° then LR_{xy} might spill, while LR_x and LR_y would not spill

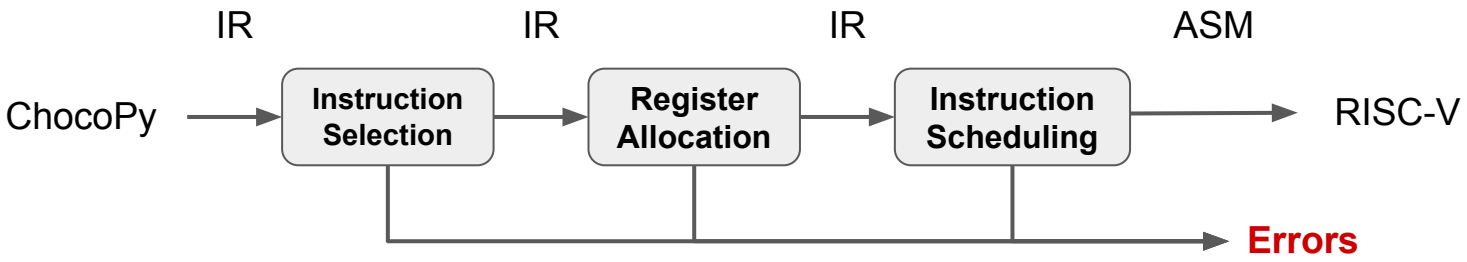


Managing Live Ranges: Coalescing

Observation led to conservative coalescing

1. Conceptually, coalesce x and y iff $x \rightarrow y \in G_I$ and $LR_{xy}{}^{\circ} < k$
2. We can do better
   - Coalesce LR_x and LR_y iff LR_{xy} has < k neighbours with degree > k
   - Only neighbours of “significant degree” can force LR_{xy} to spill
3. Always safe to perform coalesce
   - Cannot introduce a node of non-trivial degree
   - Cannot introduce a new spill

The Backend



- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces
- Automation has been less successful in the back end

Summary

- Local Allocation - spill code
- Global Allocation based on graph colouring
- Techniques to reduce spill code

Local Register Allocation

Register allocation only on basic block.

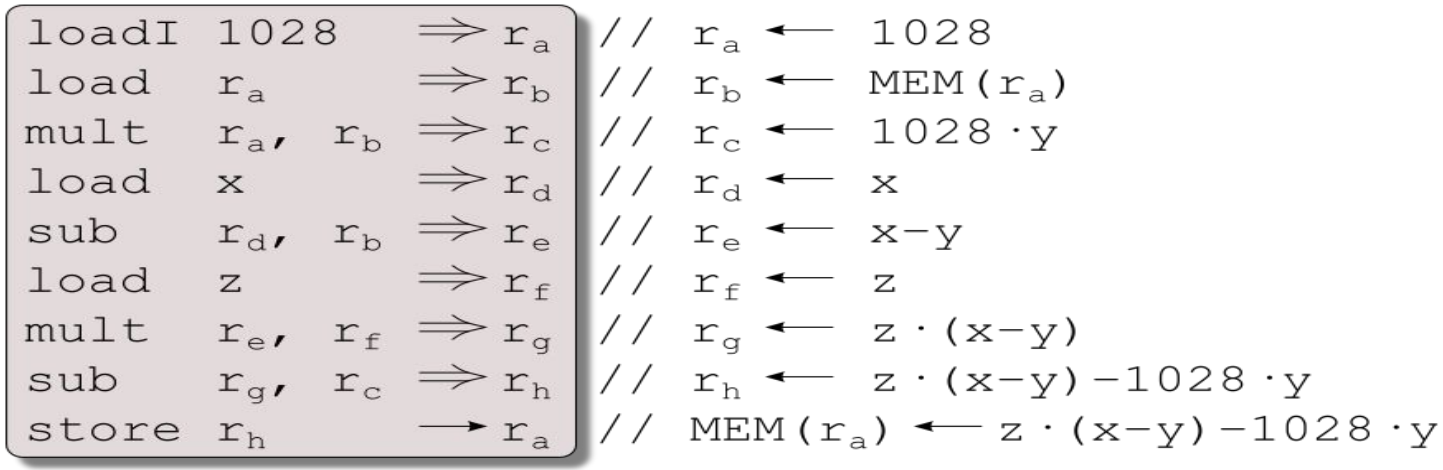
Let MAXLIVE be the maximum, over each instruction i in the block, of the number of values (pseudo-registers) live at i.

- If MAXLIVE ≤ k, allocation should be easy
- If MAXLIVE ≤ k, no need to reserve F registers for spilling
- If MAXLIVE > k, some values must be spilled to memory
- If MAXLIVE > k, need to reserve F registers for spilling

Two main forms:

- Top down
- Bottom up

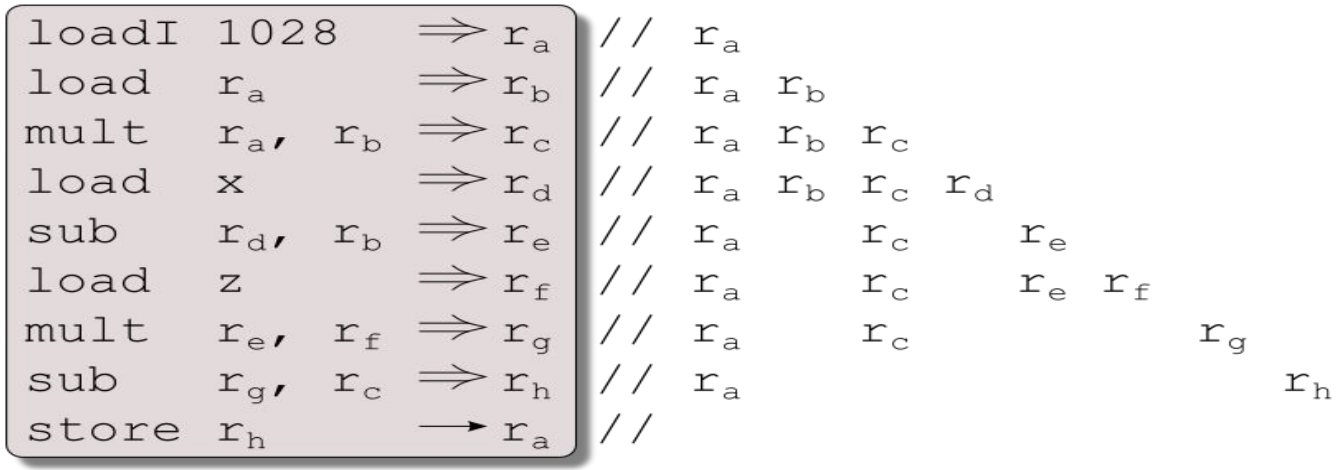
Local Register Allocation: MAXLIVE



Local Register Allocation: MAXLIVE

Example MAXLIVE computation

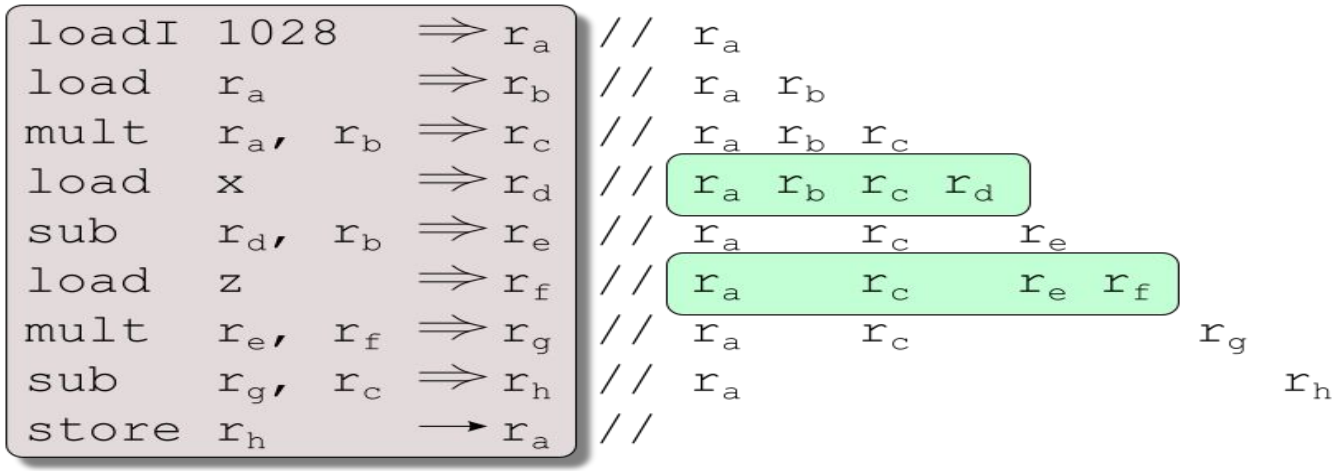
Live registers



Local Register Allocation: MAXLIVE

Example MAXLIVE computation

MAXLIVE is 4



Local register allocation: Top Down

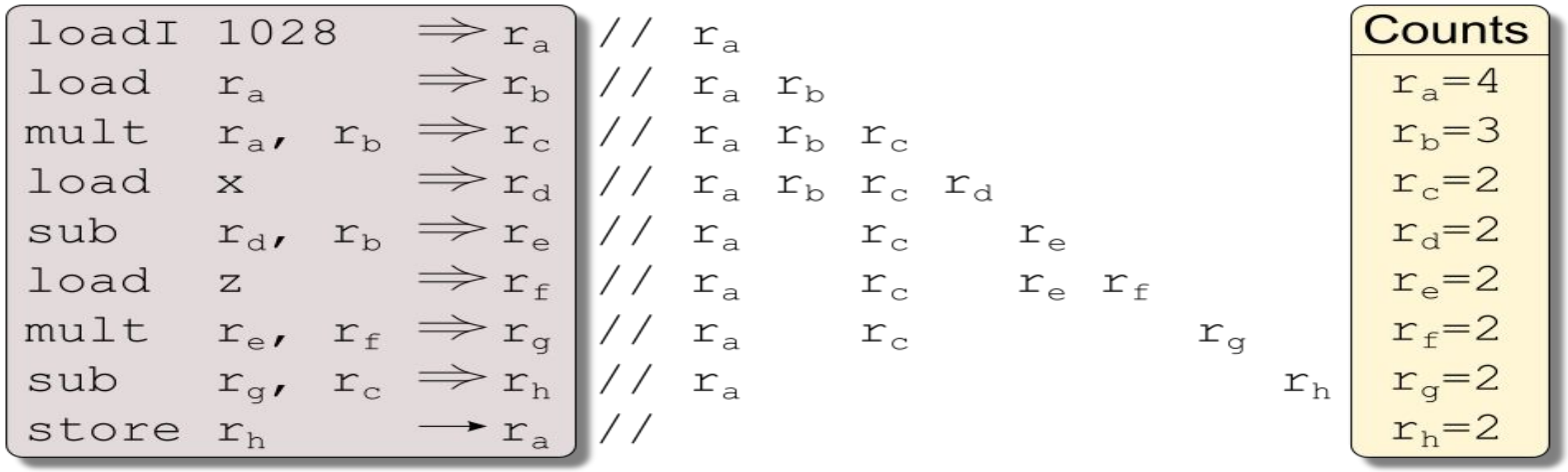
Algorithm:

- If number of values > k
  - Rank values by occurrences
  - Allocate first k - *F* values to registers
  - Spill other values

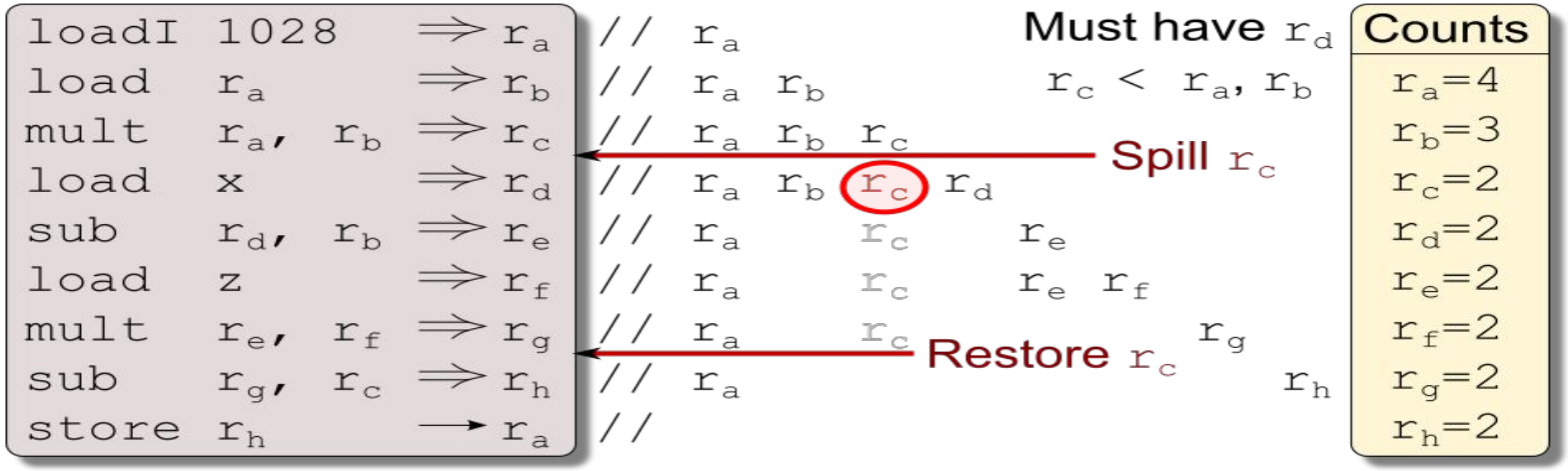
Local register allocation: top down

Example top down

Usage counts



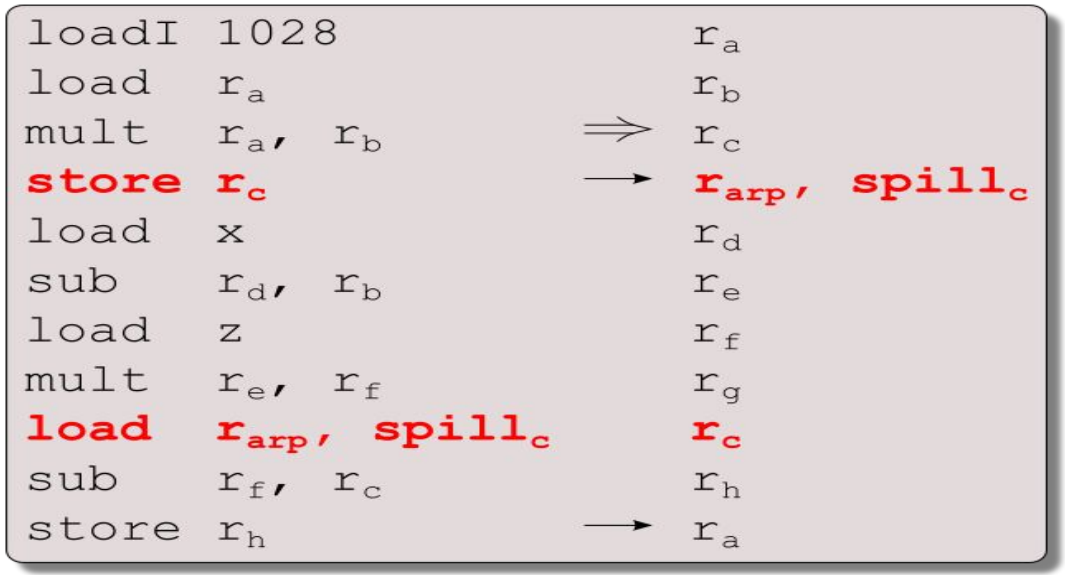
Local register allocation: Top Down



Local Register Allocation: Top down

Example top down

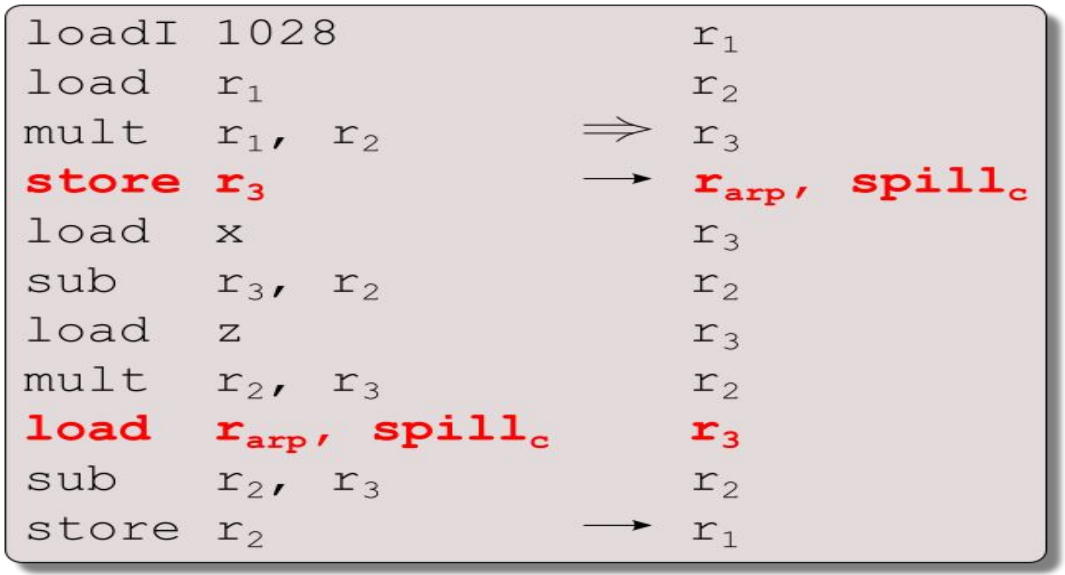
Spill code inserted



Local register allocation: Top Down

Example top down

Register assignment straightforward



Register Allocation: Alternatives to Chaitin's Algorithm

- Exhaustive allocation - go through combinatorial options - very expensive but occasional improvement
- Linear scan - fast but weak; useful for JITs

Global register allocation: Ongoing work

- Eisenbeis et al examining optimality of combined reg alloc and scheduling. Difficulty with general control-flow
- Partitioned register sets complicate matters. Allocation can require insertion of code which in turn affects allocation.
- Leupers investigated use of genetic algs for TM series partitioned reg sets.
- New work by Fabrice Rastello and others. Chordal graphs reduce complexity
- As latency increases see work in combined code generation, instruction scheduling and register allocation

Register Allocation as Coloring the Register Graph

- Degree[3] of a node (n°) is a loose upper bound on colourability
- Any node, n, such that n° < k is always trivially k-colourable
  - Trivially colourable nodes cannot adversely affect the colourability of neighbours
  - Can remove them from graph
  - Reduces degree of neighbours - may be trivially colourable
- If left with any nodes such that n° ≥ k spill one
  - Reduces degree of neighbours - may be trivially colourable

Local register allocation: Bottom Up

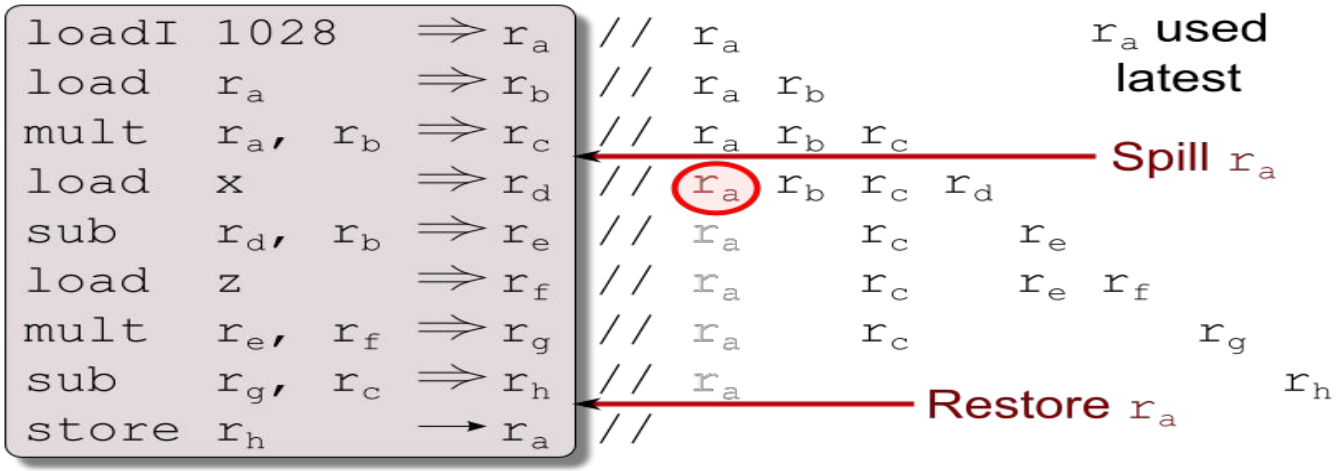
Algorithm:

- Start with empty register set
- Load on demand
- When no register is available, free one

Replacement:

- Spill the value whose next use is farthest in the future
- Prefer clean value to dirty value

Local register allocation: Bottom Up



Local register allocation: Bottom Up

Example bottom up

Spill code inserted