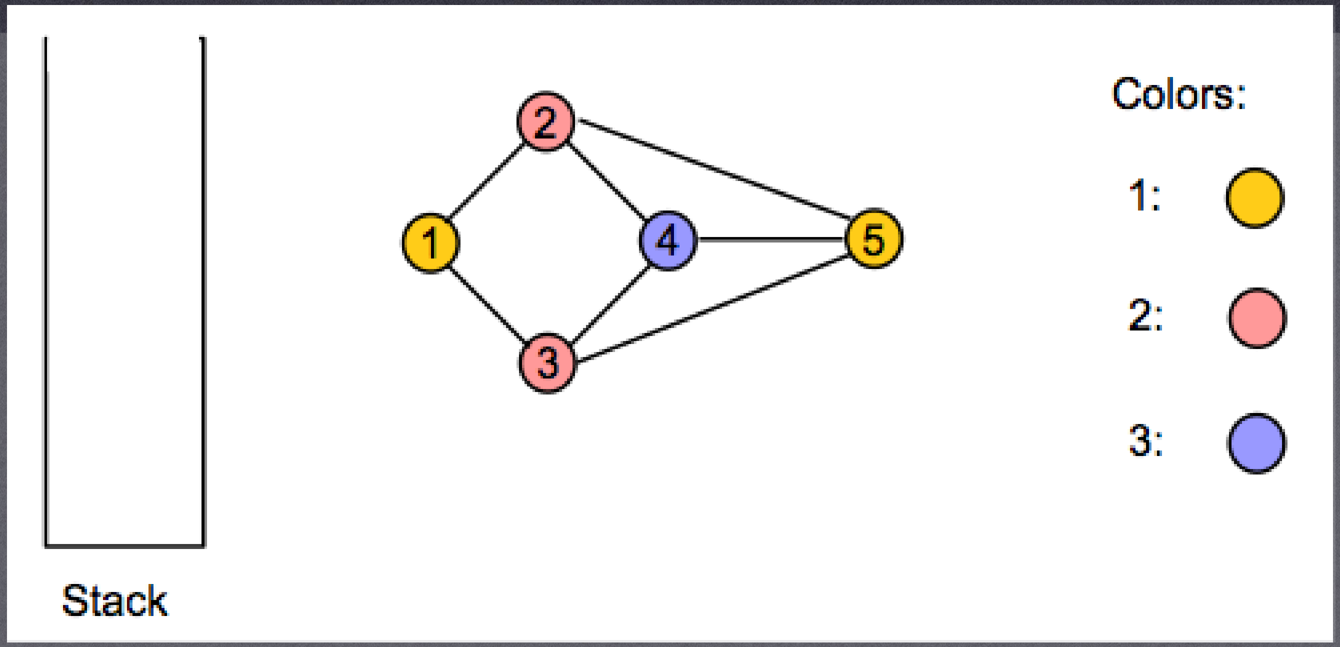


Compiling Techniques

Lecture 15: Register Allocation

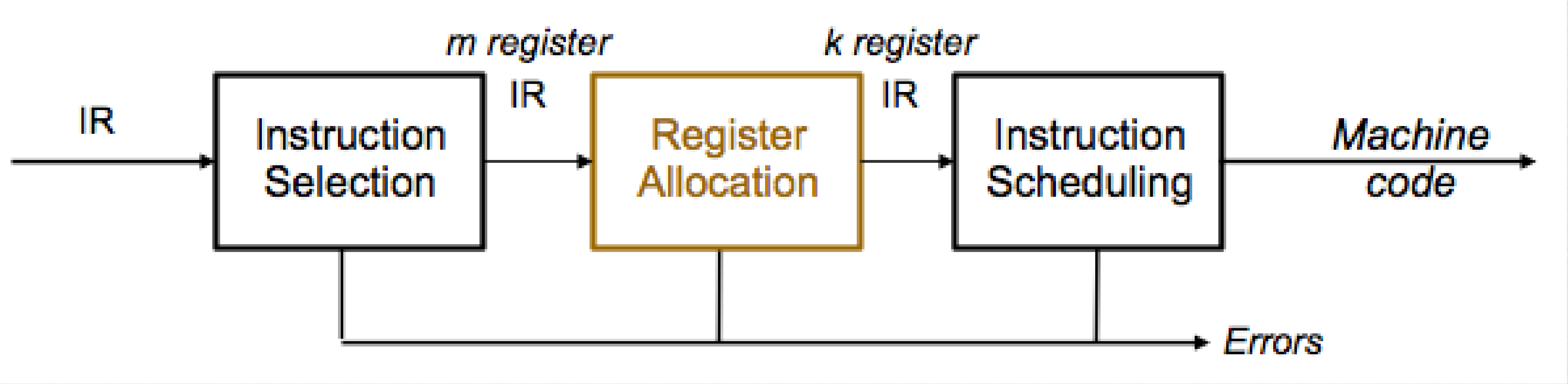
Christophe Dubach

EaC : Chapter 13

Overview

- Data Flow Analysis
- Local Register Allocation
- Global Register Allocation via Graph Colouring

Register Allocation



- Critical properties
  - 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 log n), maybe O(n*n), but not O(exp(n))

Register Allocation

- The Task
  - At each point in the code, pick the values to keep in registers
  - Insert code to move values between registers & memory
  - Minimise inserted code
  - Make good use of any extra registers
- Allocation versus assignment
  - Allocation is deciding which values to keep in registers
  - Assignment is choosing specific registers for values
  - This distinction is often lost in the literature
  - The compiler must perform both allocation & assignment

Basic Blocks

- Definition
  - **A basic block is a maximal length segment of straight-line (i.e., branch free) code**
- Importance (assuming normal execution)
  - Strongest facts are provable for branch-free code
  - If any statement executes, they all execute
  - Execution is totally ordered
- Optimisation
  - Many techniques for improving basic blocks
  - Simplest problems
  - Strongest methods

Data Flow Analysis

- Idea
  - Data-flow analysis derives information about the dynamic behaviour of a program by only examining the static code
- Example
  - How many registers do we need for the program below?
  - Easy bound: the number of variables used (3)
  - Better answer is found by considering the dynamic requirements of the program

```
     a := 0
L1:  b := a + 1
     c := c + b
     a := b *2
     if a < 9 goto L1
     return c
```

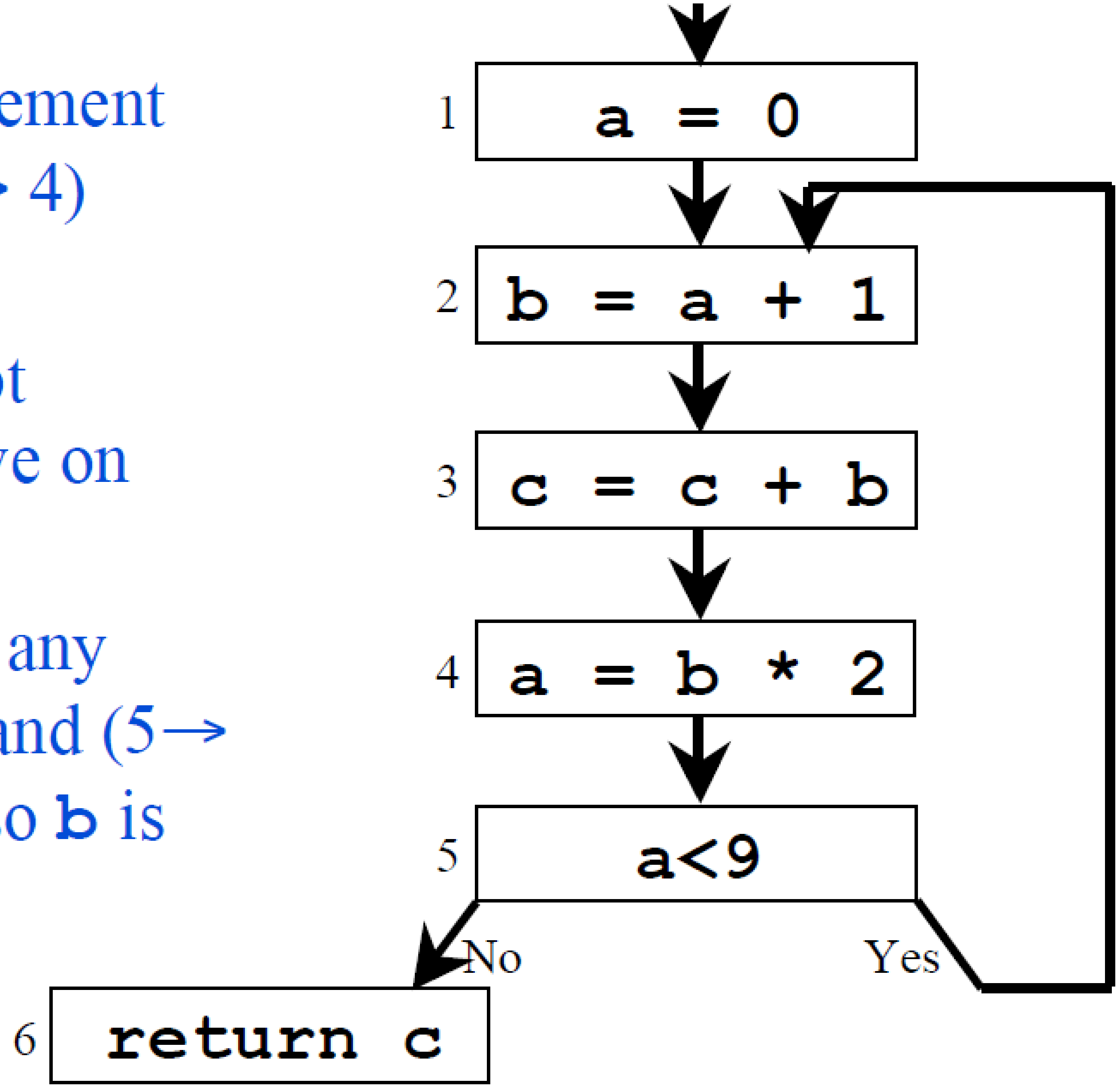
Liveness Analysis

- Definition
  - A variable is live at a particular point in the program if its value at that point will be used in the future (dead, otherwise).
  - To compute liveness at a given point, we need to look into the future
- Motivation: Register Allocation
  - A program contains an unbounded number of variables
  - Must execute on a machine with a bounded number of registers
  - Two variables can use the same register if they are never in use at the same time (i.e, never simultaneously live).
  - Register allocation uses liveness information

Example

What is the live range of `b`?

- Variable **`b`** is read in statement 4, so **`b`** is live on the (3 → 4) edge
- Since statement 3 does not assign into **`b`**, **`b`** is also live on the (2→3) edge
- Statement 2 assigns **`b`**, so any value of **`b`** on the (1→2) and (5→ 2) edges are not needed, so **`b`** is dead along these edges



`b`'s live range is (2→3→4)

Example Continued

Live range of a

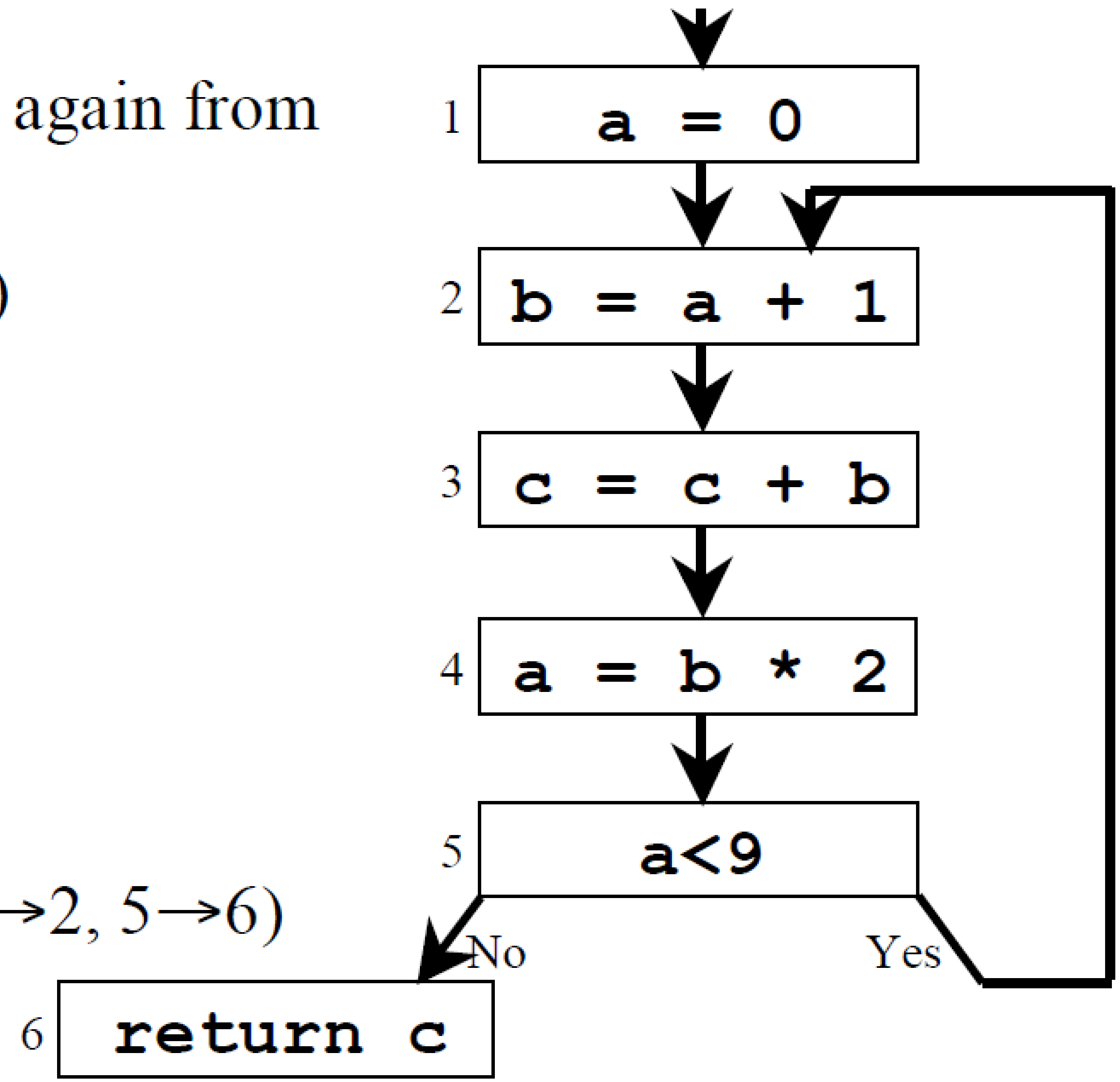
- a is live from (1→2) and again from (4→5→2)
- a is dead from (2→3→4)

Live range of b

- b is live from (2→3→4)

Live range of c

- c is live from (entry→1→2→3→4→5→2, 5→6)

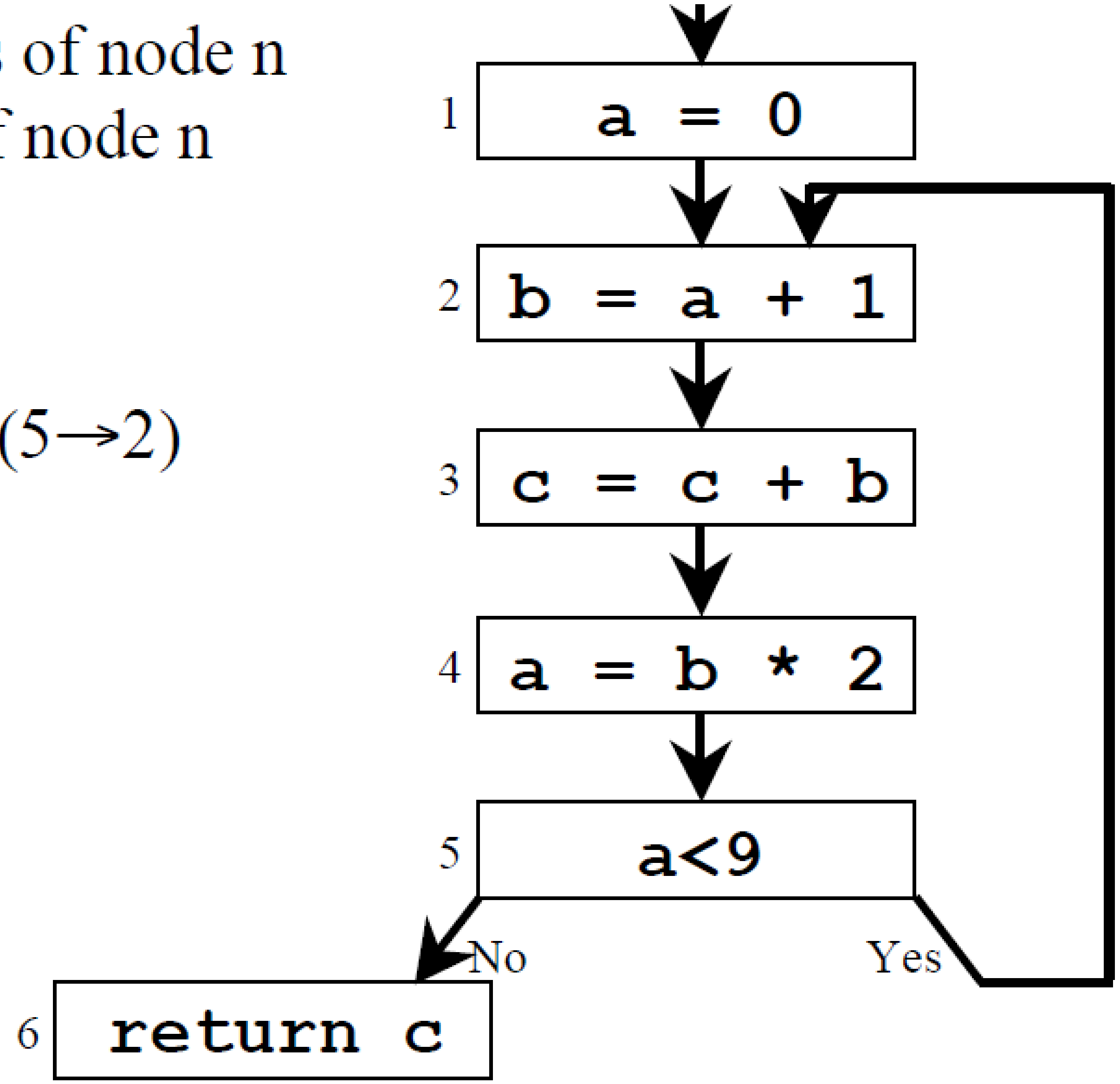


Variables a and b are never simultaneously live, so they can share a register

Terminology

Flow Graph Terms

- A CFG node has **out-edges** that lead to **successor** nodes and **in-edges** that come from **predecessor** nodes
- **pred[n]** is the set of all predecessors of node n
  succ[n] is the set of all successors of node n



Examples

- Out-edges of node 5: (5→6) and (5→2)
- succ[5] = {2,6}
- pred[5] = {4}
- pred[2] = {1,5}

Uses and Defs

Def (or definition)

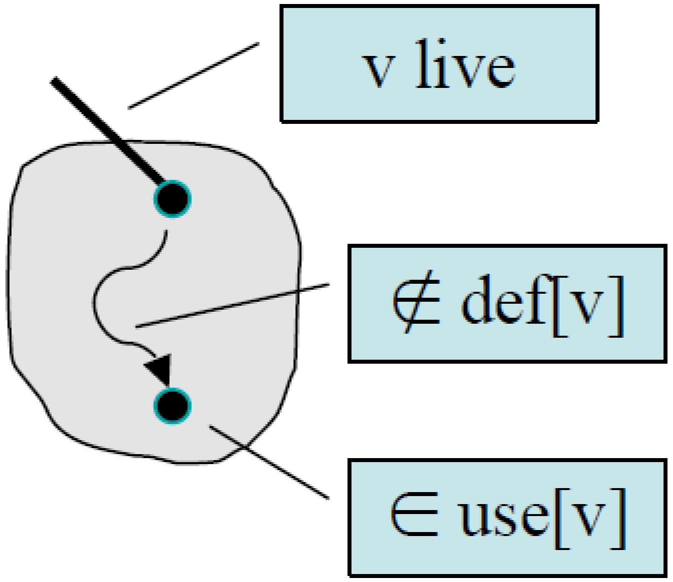
- An **assignment** of a value to a variable
- def[v] = set of CFG nodes that define variable v
- def[n] = set of variables that are defined at node n

```
a = 0
```

```
a < 9?
```

Use

- A **read** of a variable's value
- use[v] = set of CFG nodes that use variable v
- use[n] = set of variables that are used at node n



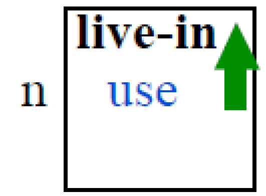
More precise definition of liveness

- A variable v is live on a CFG edge if
  (1) ∃ a directed path from that edge to a use of v (node in use[v]), and
  (2) that path does not go through any def of v (no nodes in def[v])

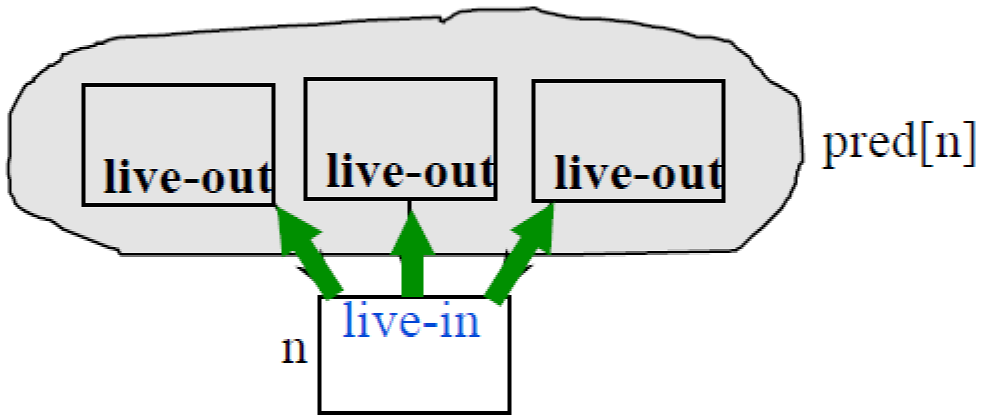
Computing Liveness

Rules for computing liveness

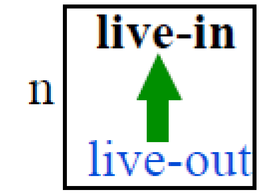
(1) Generate liveness:
If a variable is in use[n],
it is live-in at node n



(2) Push liveness across edges:
If a variable is live-in at a node n
then it is live-out at all nodes in pred[n]



(3) Push liveness across nodes:
If a variable is live-out at node n and not in def[n]
then the variable is also live-in at n



Data-flow equations

$$\text{(1)}\ \ \mathbf{in[n] = use[n]} \cup \mathbf{(out[n] - def[n])}\ \ \text{(3)}$$

$$\mathbf{out[n]} = \bigcup_{s \in succ[n]} \mathbf{in[s]}\ \ \text{(2)}$$

FIX-POINT ALGORITHM

Local Register Allocation

- What's "local" ? (as opposed to "global")
  - A local transformation operates on basic blocks
  - Many optimisations are done locally
- Does local allocation solve the problem?
  - It produces decent register use inside a block
  - Inefficiencies can arise at boundaries between blocks
- How many passes can the allocator make?
  - This is an off-line problem
  - As many passes as it takes

Observations

- Allocator may need to reserve registers to ensure feasibility
  - Must be able to compute addresses
  - Requires some minimal set of registers, F
    - F depends on target architecture
  - Use these registers only for spilling
- What if k–F < |values| < k?
  - Check for this situation
  - Adopt a more complex strategy (iterate?)
  - Accept the fact that the technique is an approximation
- |values| > k?
  - Some values must be spilled to memory

Top-down Versus Bottom-up Allocation

- Top-down allocator
  - Work from external notion of what is important
  - Assign registers in priority order
  - Save some registers for the values relegated to memory
- Bottom-up allocator
  - Work from detailed knowledge about problem instance
  - Incorporate knowledge of partial solution at each step
  - Handle all values uniformly

Top-down Allocator

- The idea:
  - Keep busiest values in a register
  - Use the reserved set, F, for the rest
- Algorithm:
  - Rank values by number of occurrences
  - Allocate first k – F values to registers
  - Rewrite code to reflect these choices
- Common technique of 60’s and 70’s

Bottom-up Allocator

- The idea:
  - Focus on replacement rather than allocation
  - Keep values used “soon” in registers
- 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
  - Sound familiar? Think page replacement ...

Example

```
loadI   1028     ⇒ r1    // r1 ← 1028
load    r1       ⇒ r2    // r2 ← MEM(r1) == y
mult    r1, r2   ⇒ r3    // r3 ← 2 · y
loadI   x        ⇒ r4    // r4 ← x
sub     r4, r2   ⇒ r5    // r5 ← x - y
loadI   z        ⇒ r6    // r6 ← z
mult    r5, r6   ⇒ r7    // r7 ← z · (x - y)
sub     r7, r3   ⇒ r8    // r5 ← z · (x - y) - (2 · y)
store   r8       ⇒ r1    // MEM(r1) ← z · (x - y) - (2 · y)
```

Live Ranges

```
loadI   1028    ⇒ r1   // r1
load    r1      ⇒ r2   // r1 r2
mult    r1, r2  ⇒ r3   // r1 r2 r3
loadI   x       ⇒ r4   // r1 r2 r3 r4
sub     r4, r2  ⇒ r5   // r1    r3    r5
loadI   z       ⇒ r6   // r1    r3    r5 r6
mult    r5, r6  ⇒ r7   // r1    r3          r7
sub     r7, r3  ⇒ r8   // r1                   r8
store   r8      ⇒ r1   //
```

Top Down (3 Regs)

```
loadI   1028     ⇒ r1   // r1
load    r1       ⇒ r2   // r1 r2
mult    r1, r2   ⇒ r3   // r1 r2 r3
loadI   x        ⇒ r4   // r1 r2 r3 r4
sub     r4, r2   ⇒ r5   // r1    r3    r5
loadI   z        ⇒ r6   // r1    r3    r5 r6
mult    r5, r6   ⇒ r7   // r1    r3          r7
sub     r7, r3   ⇒ r8   // r1                   r8
store   r8       ⇒ r1   //
```

R3 LEAST FREQUENTLY USED

Bottom Up (3 Regs)

```
loadI   1028    ⇒ r1    // r1
load    r1      ⇒ r2    // r1 r2
mult    r1, r2  ⇒ r3    // r1 r2 r3
loadI   x       ⇒ r4    // r1 r2 r3 r4   >3 REGISTERS
sub     r4, r2  ⇒ r5    // r1    r3    r5
loadI   z       ⇒ r6    // r1    r3    r5 r6
mult    r5, r6  ⇒ r7    // r1    r3          r7
sub     r7, r3  ⇒ r8    // r1                   r8
store   r8      ⇒ r1    //
```

R1 USE FARTHEST AWAY

Graph Colouring Register Allocation

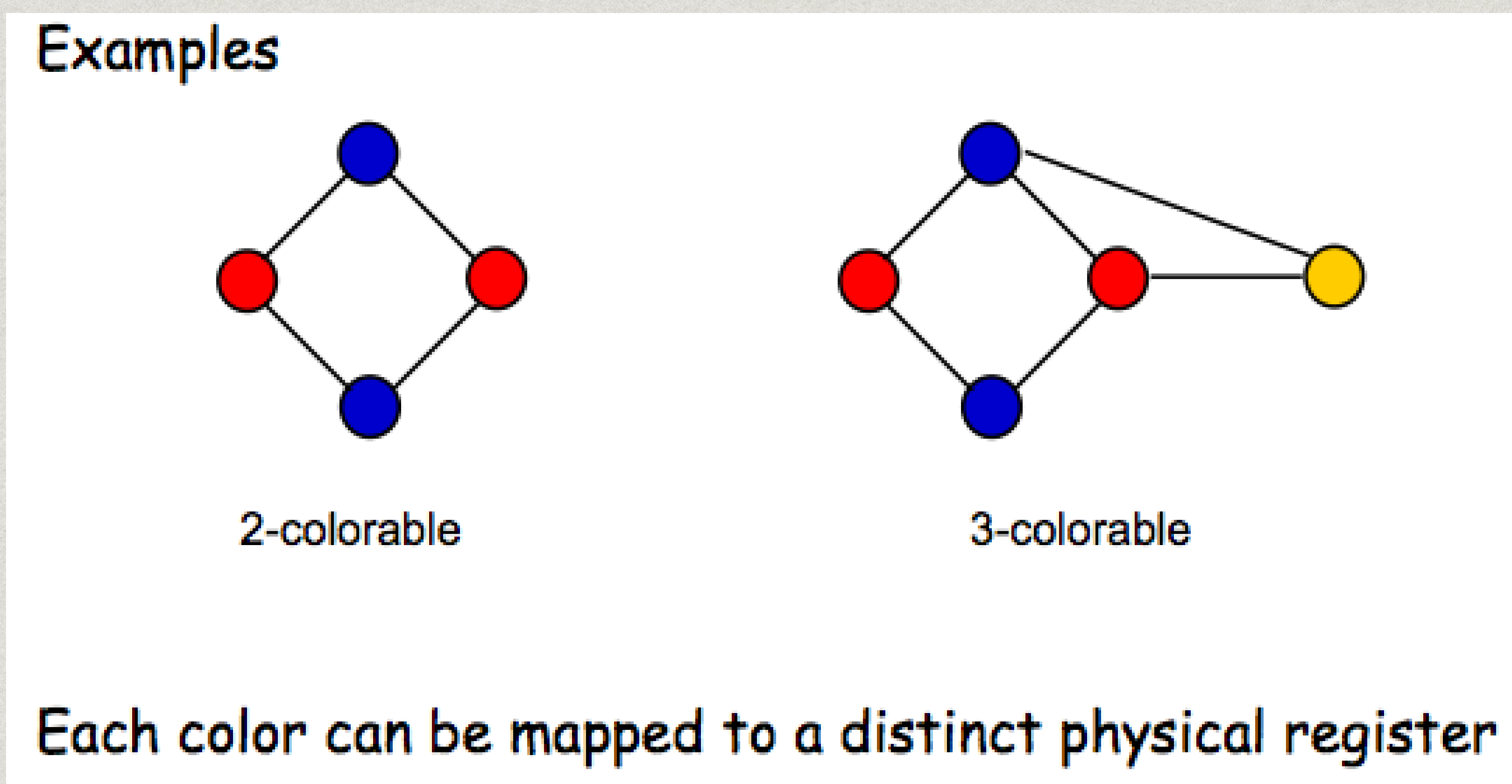
- Idea:

- Build a “conflict graph” or “interference graph”
  - Nodes - Virtual Registers
  - Edges - Overlapping Live Ranges

- Find a k-colouring for the graph, or change the code to a nearby problem that it can k-colour
  - Colours - Physical Registers

Graph Colouring

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



Interference Graph

- What is an “interference” ? (or conflict)
  - Two values interfere if there exists an operation where both are simultaneously live
  - If x and y interfere, they cannot occupy the same register
- To compute interferences, we must know where values are “live”
- Interference graph G_I
  - Nodes in G_I represent values, or live ranges
  - Edges in G_I represent individual interferences
    - For $x,y \in G_I$, $(x,y) \in G_I$ iff x and y interfere
  - A k-colouring of G_I can be mapped into an allocation to k registers

Observations

- Suppose you have k registers
  - Look for a k colouring
- Any vertex n that has fewer than k neighbours in the interference graph($n^\circ < k$) can always be coloured !
- Pick any colour not used by its neighbours — there must be one

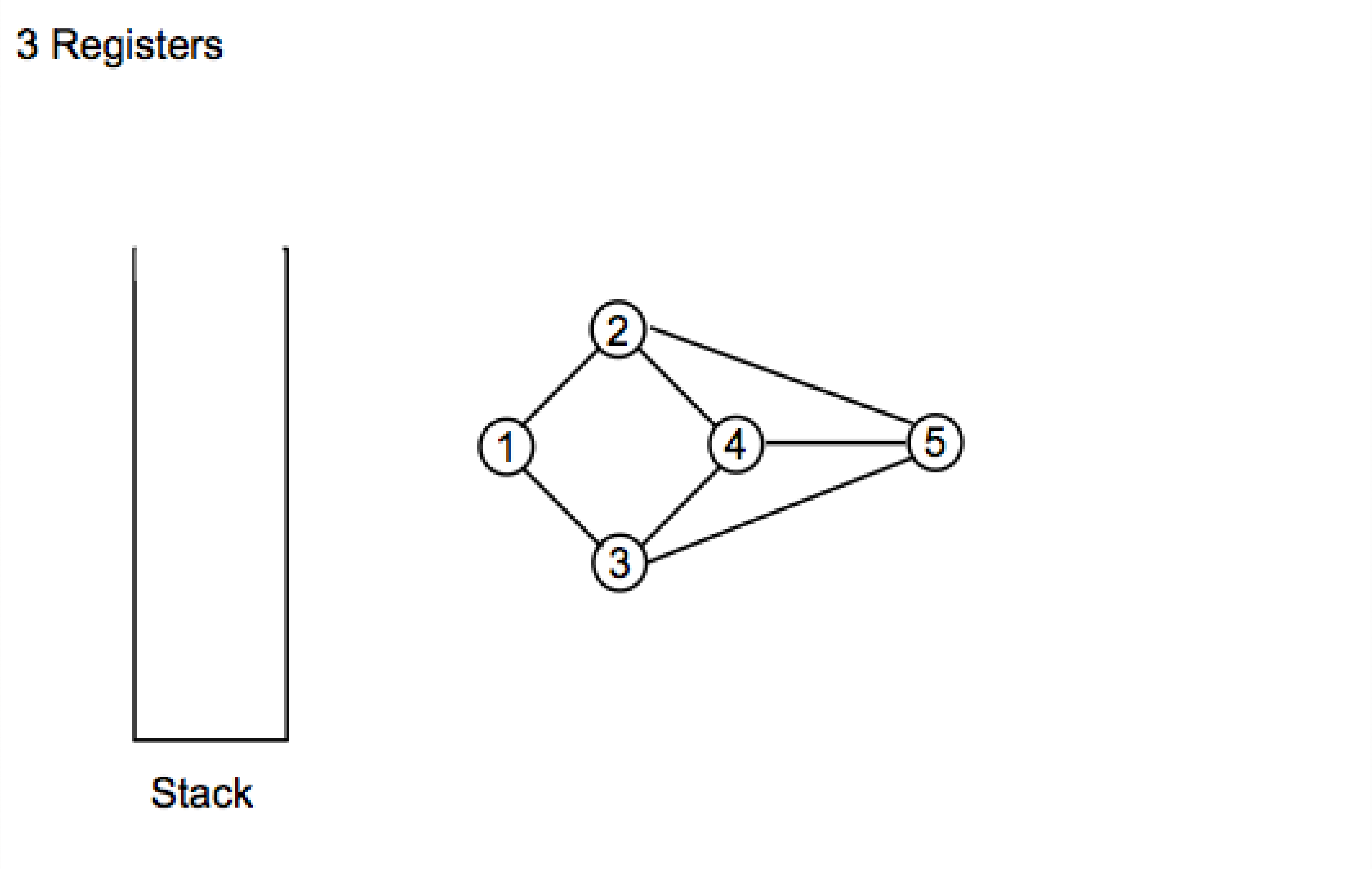
Ideas behind algorithm

- Pick any vertex n such that n°< k and put it on the stack
- Remove that vertex and all edges incident from the interference graph
  - This may make some new nodes have fewer than k neighbours
- At the end, if some vertex n still has k or more neighbours, then spill the live range associated with n
- Otherwise successively pop vertices off the stack and colour them in the lowest colour not used by some neighbour

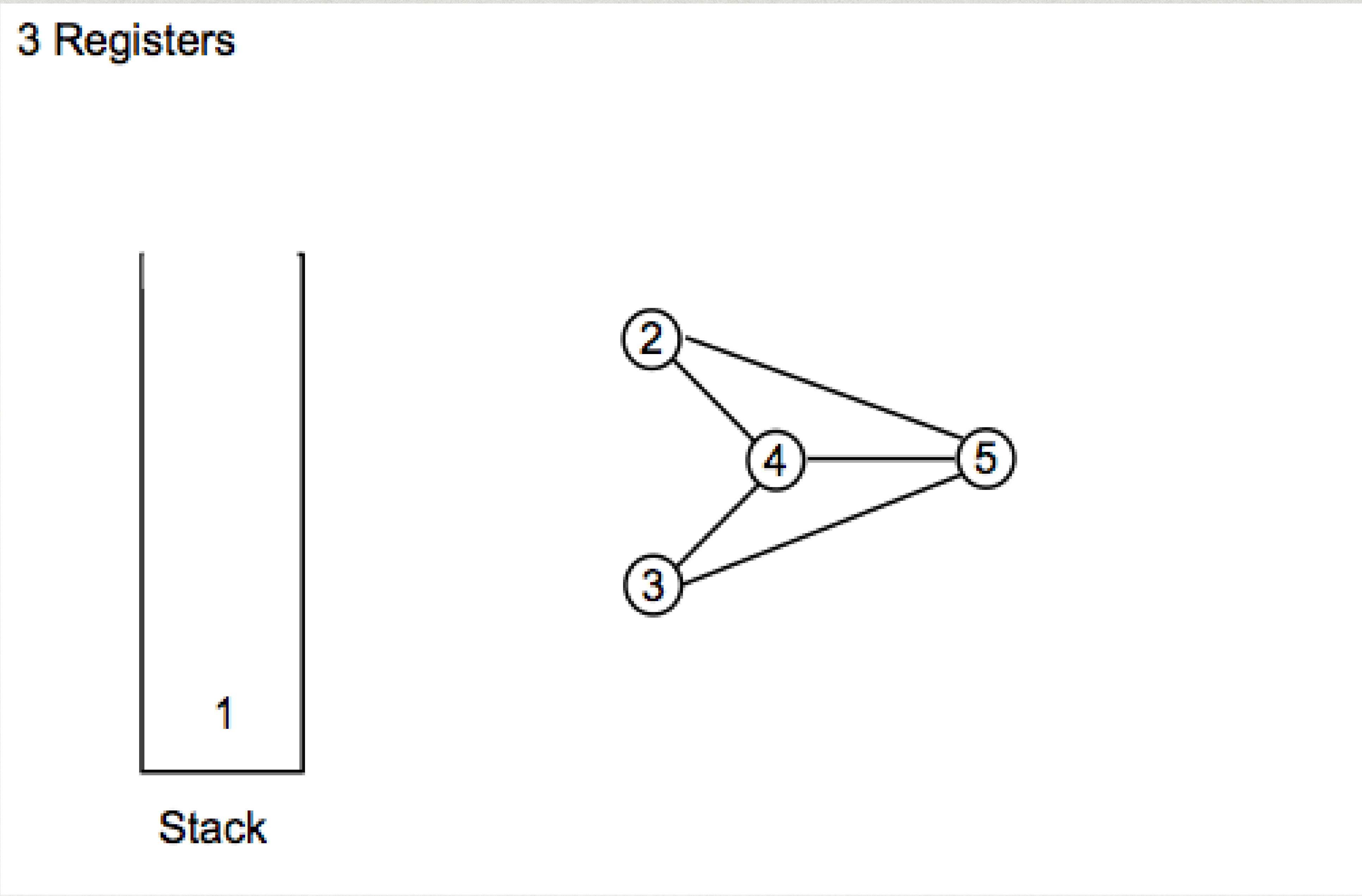
Chaitin's Algorithm

- While ∃ vertices with $<k$ neighbours in G_I
  - Pick any vertex n such that $n^\circ < k$ and put it on the stack
  - Remove that vertex and all edges incident to it from G_I
  - This will lower the degree of n's neighbours
- If G_I is non-empty (all vertices have k or more neighbours) then:
  - Pick a vertex n (using some heuristic) and spill the live range associated with n
  - Remove vertex n from G_I, along with all edges incident to it and put it on the stack
  - If this causes some vertex in G_I to have fewer than k neighbours, then go to step 1; otherwise, repeat step 2
- Successively pop vertices off the stack and colour them in the lowest colour not used by some neighbour

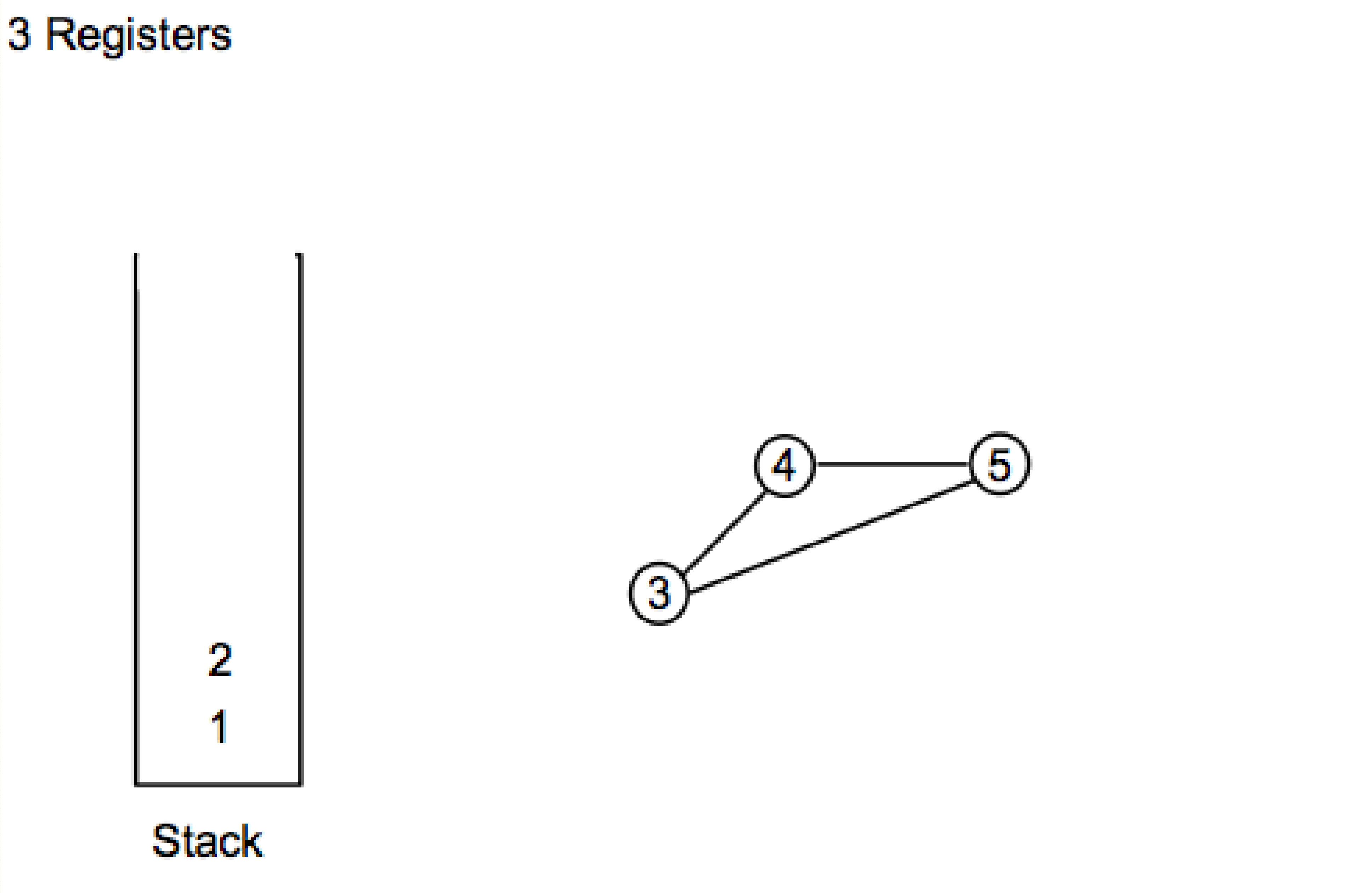
Example (3 Registers)



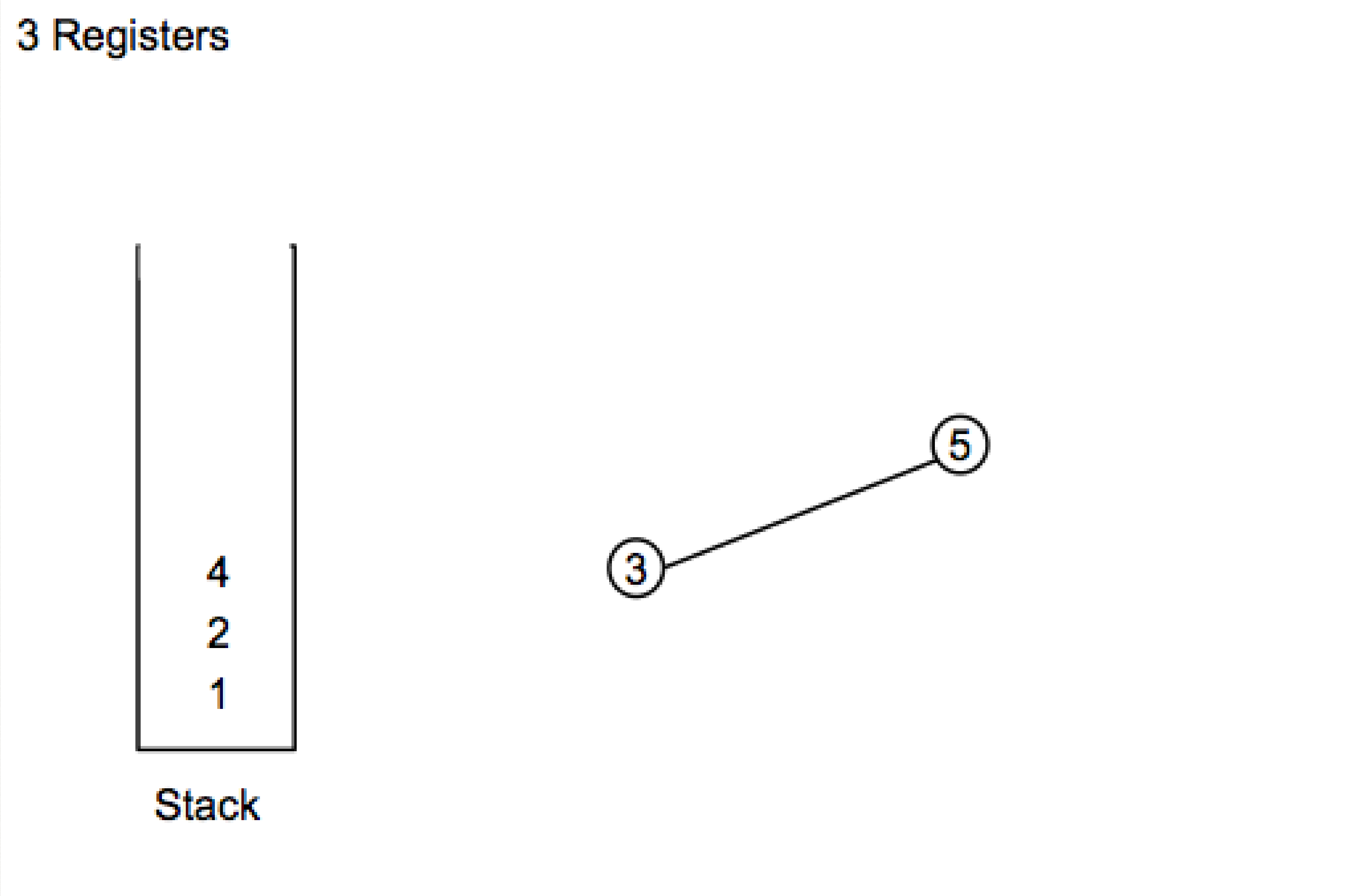
Example (3 Registers)



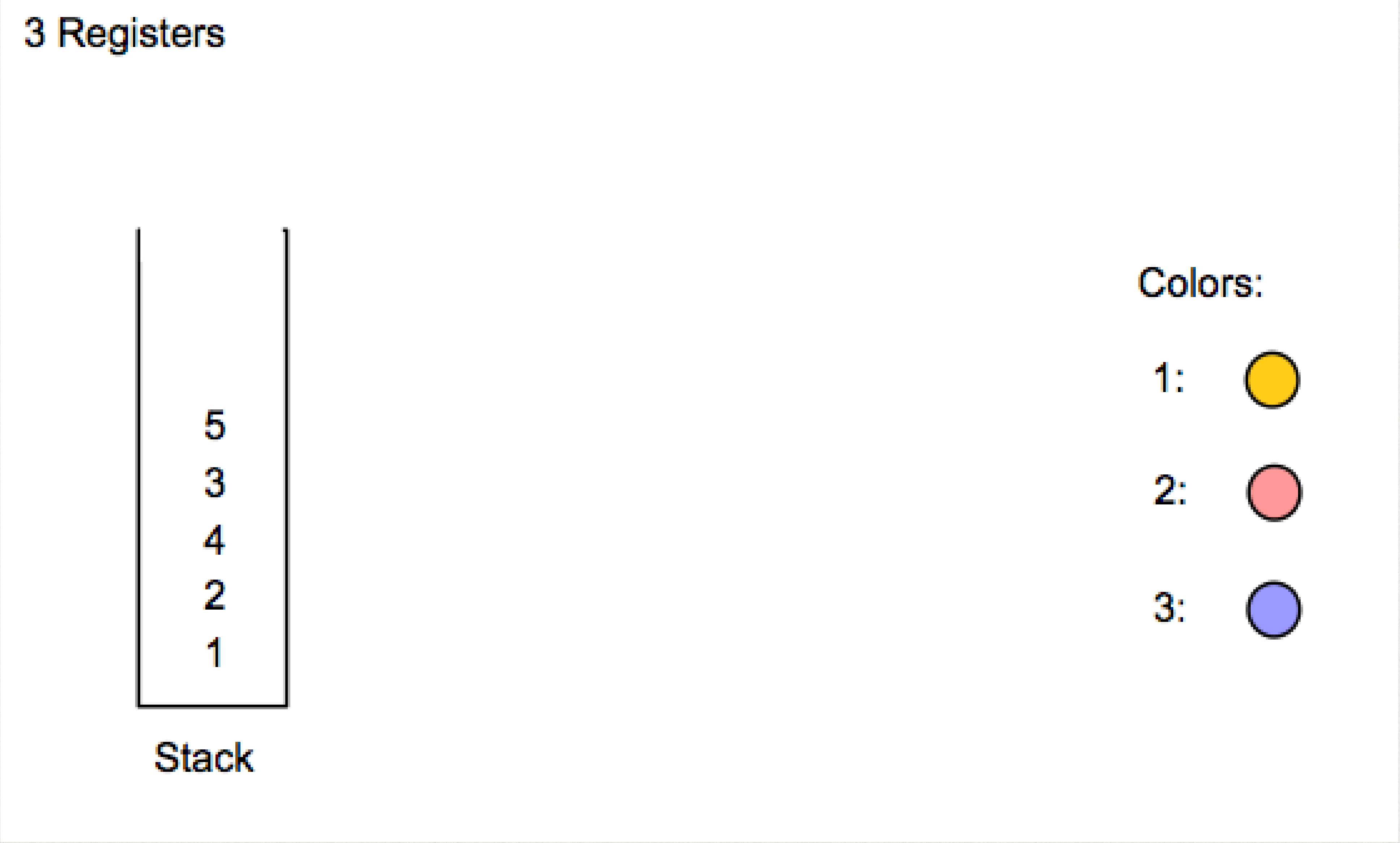
Example (3 Registers)



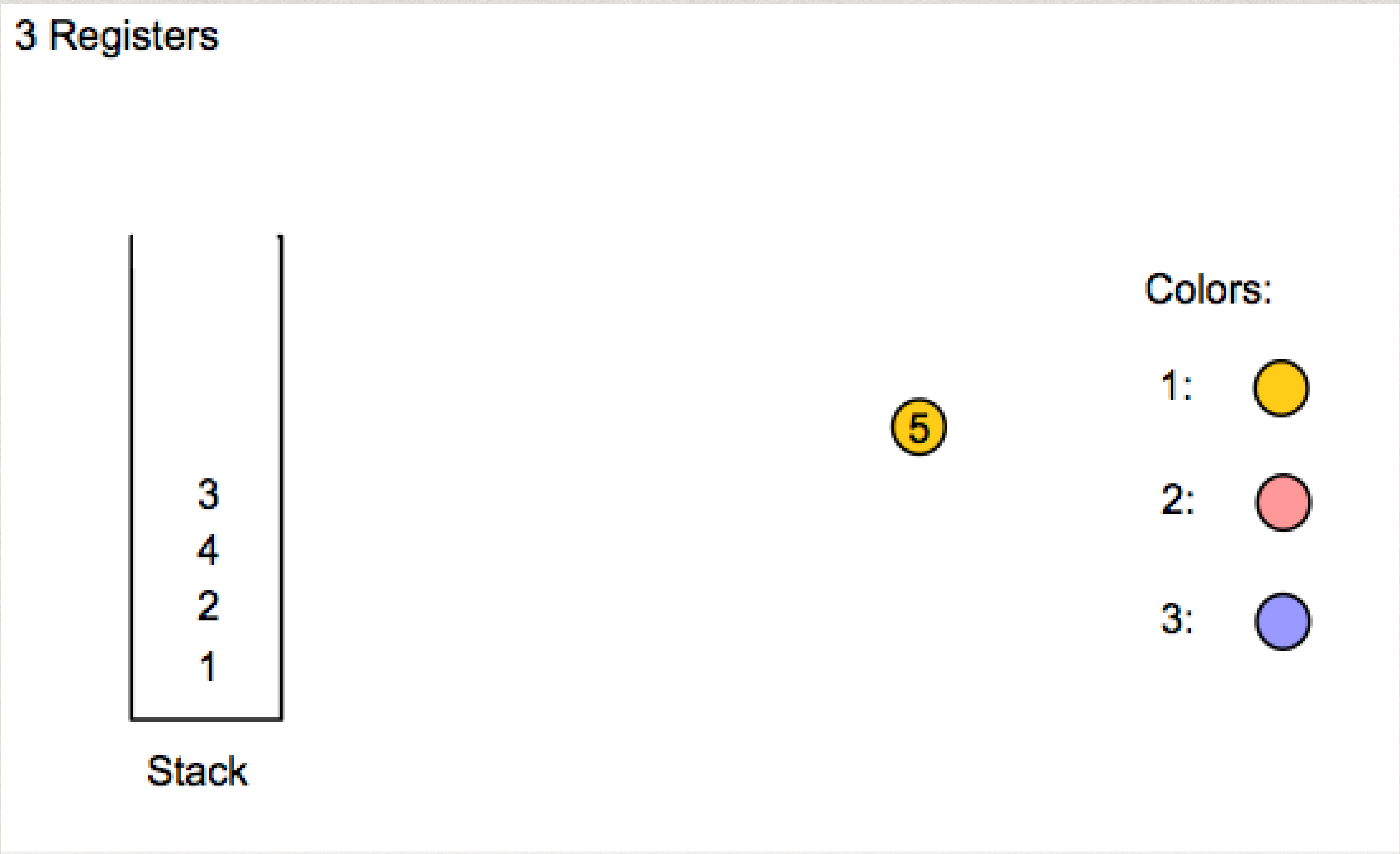
Example (3 Registers)



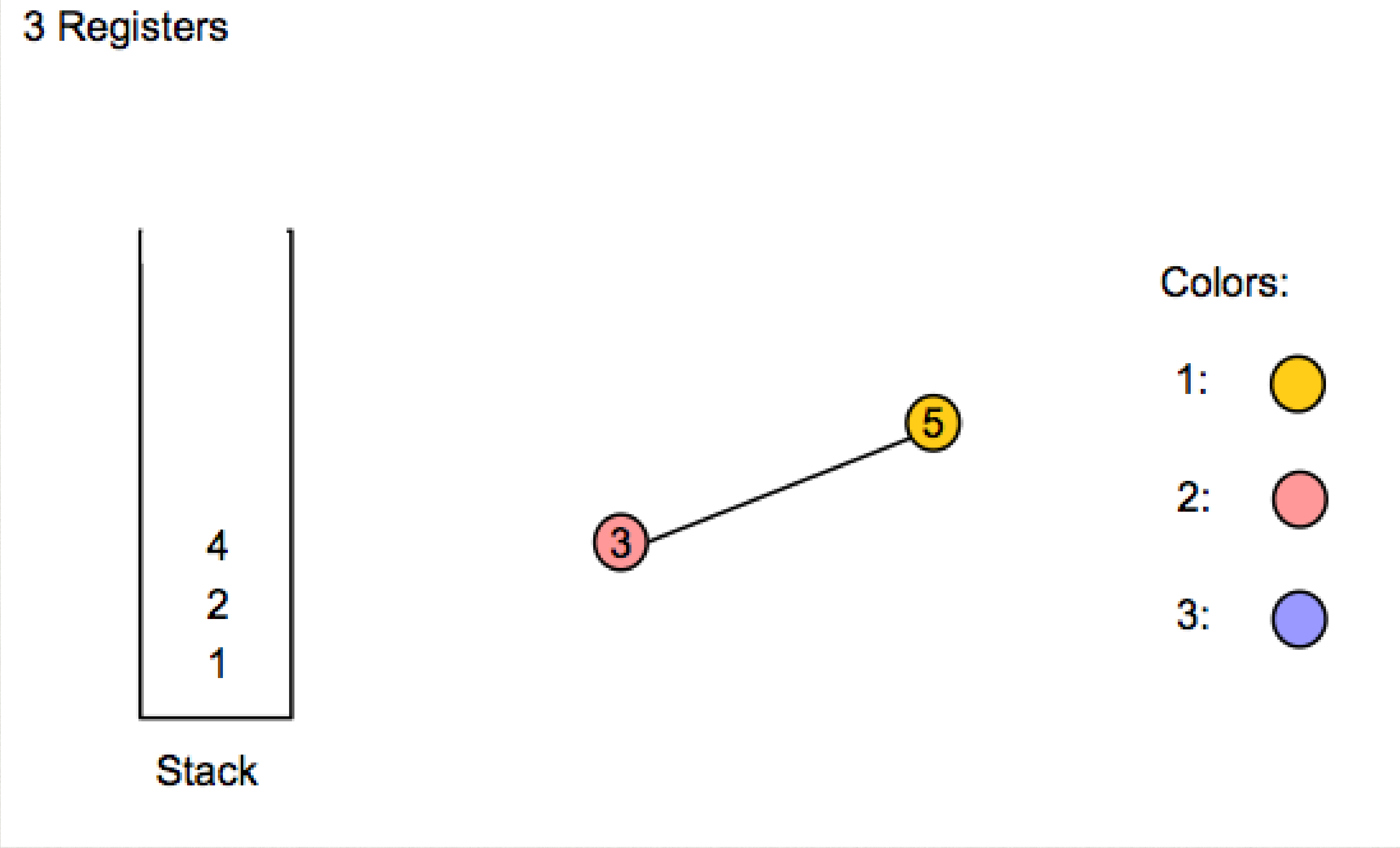
Example (3 Registers)



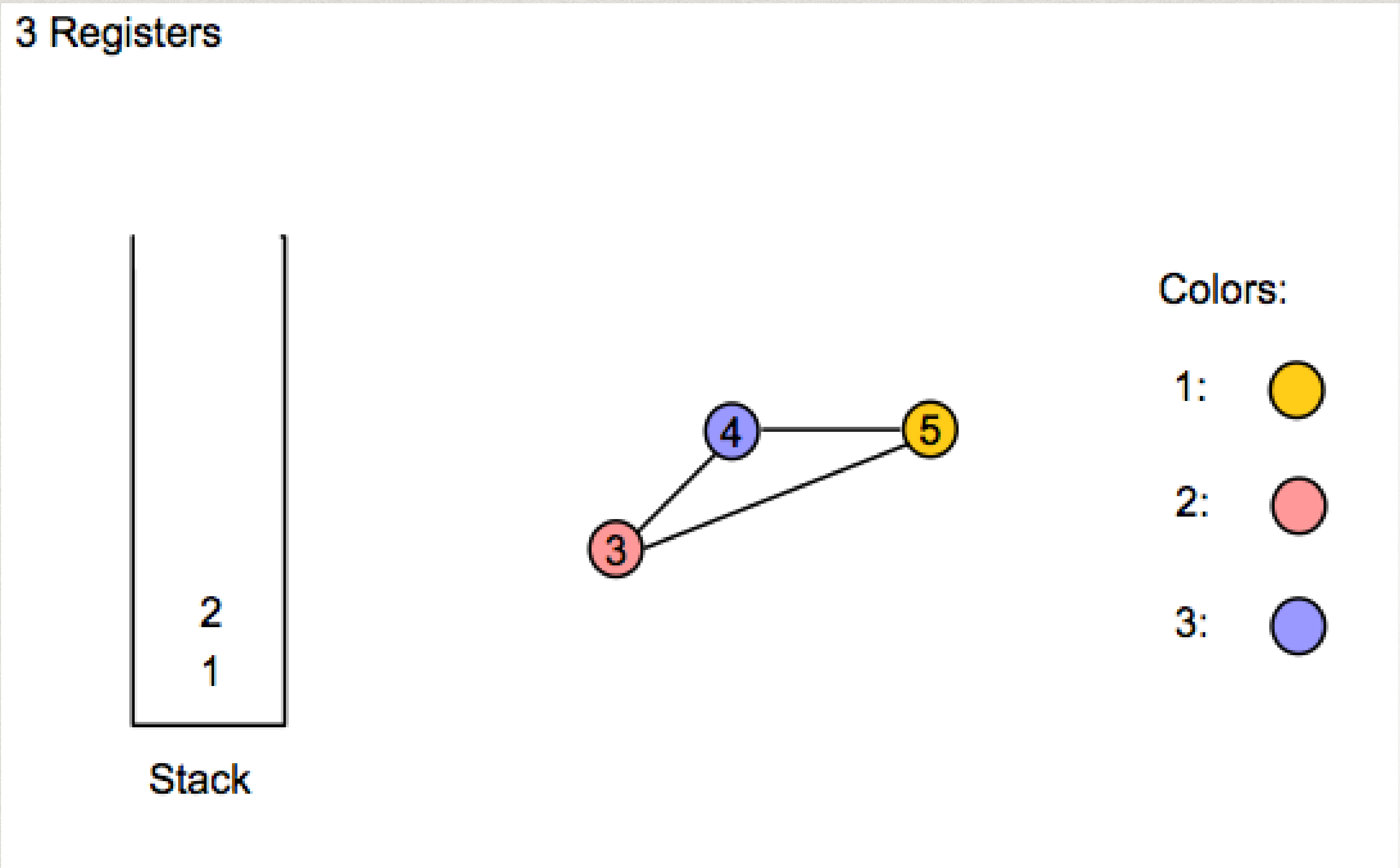
Example (3 Registers)



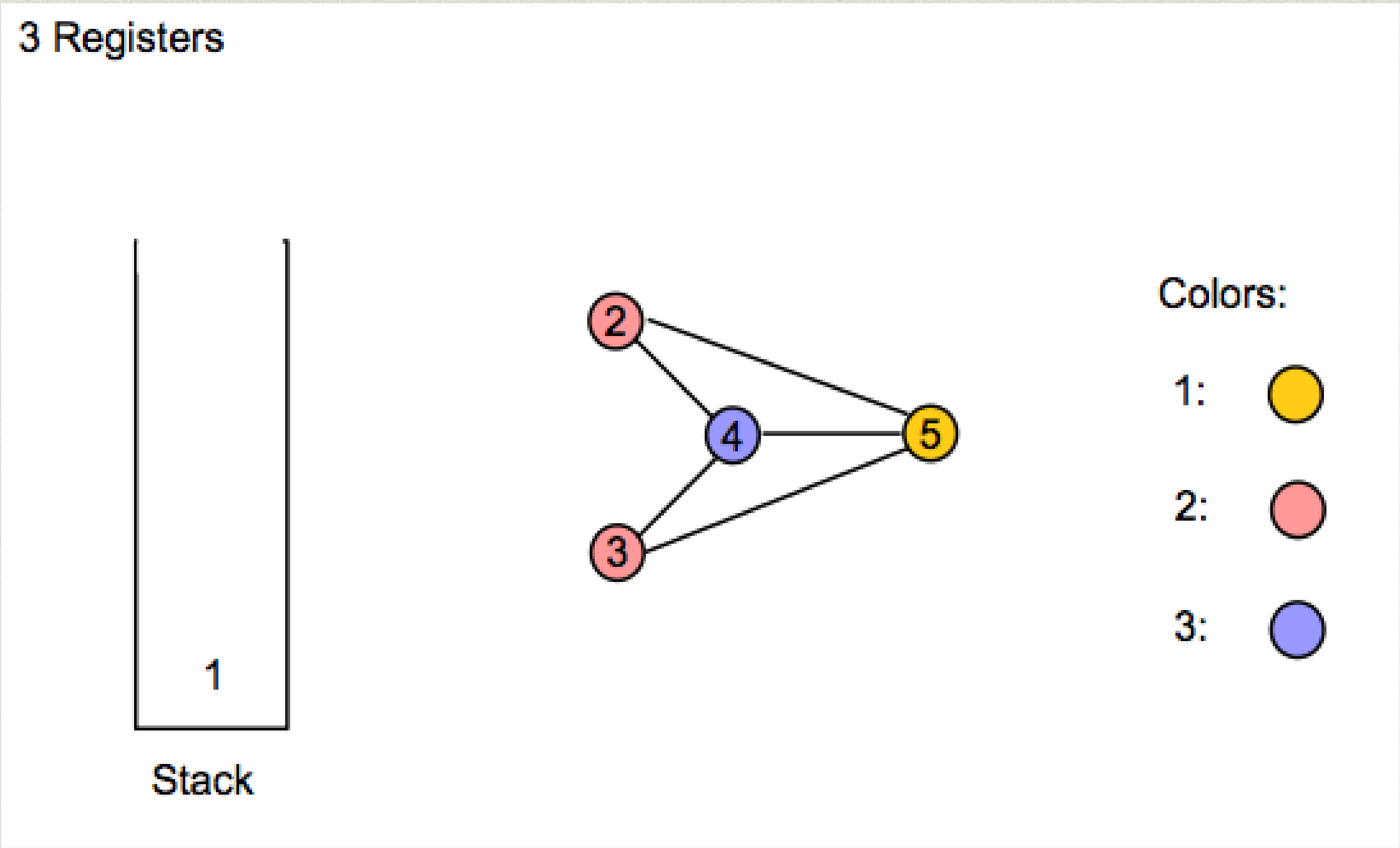
Example (3 Registers)



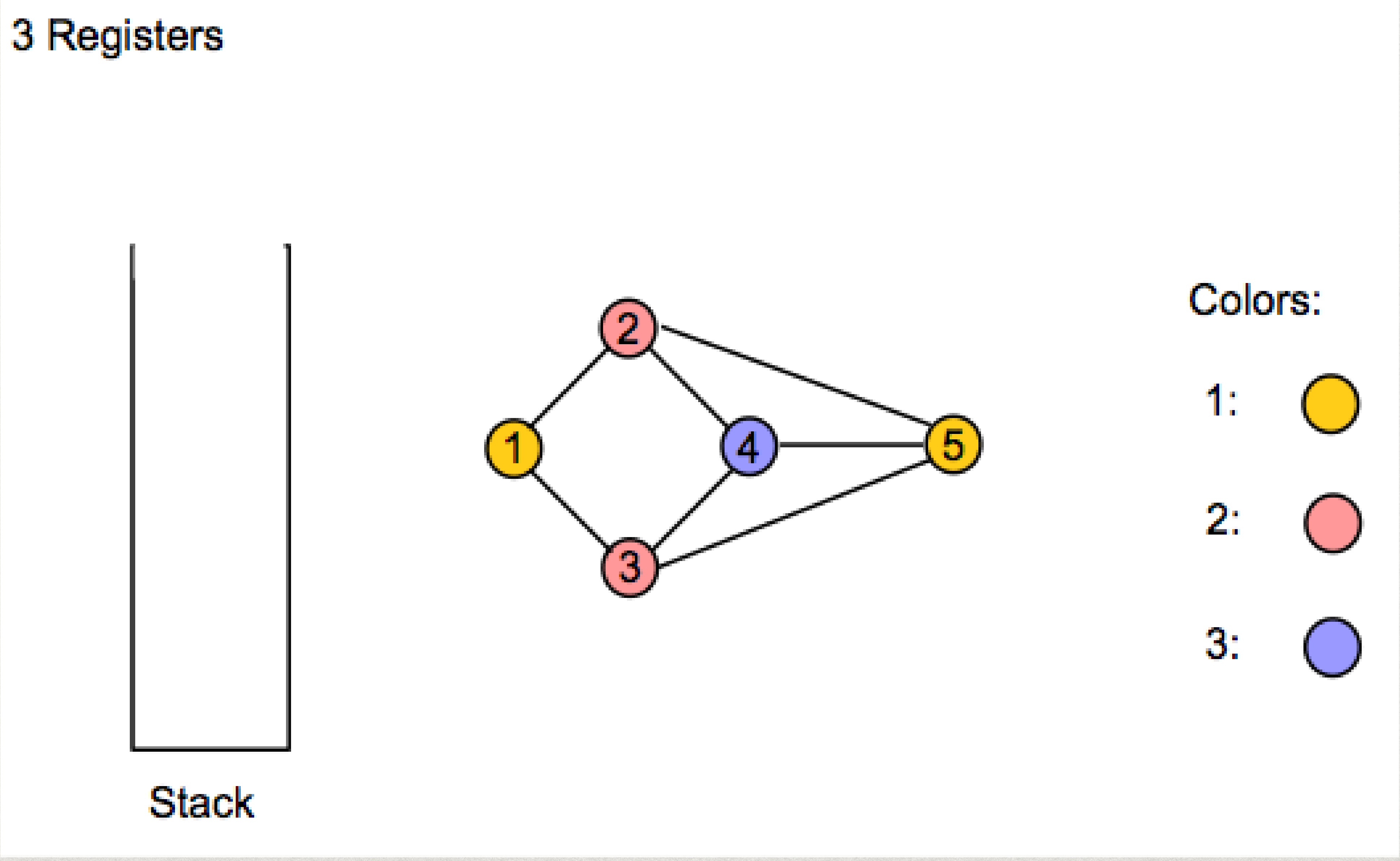
Example (3 Registers)



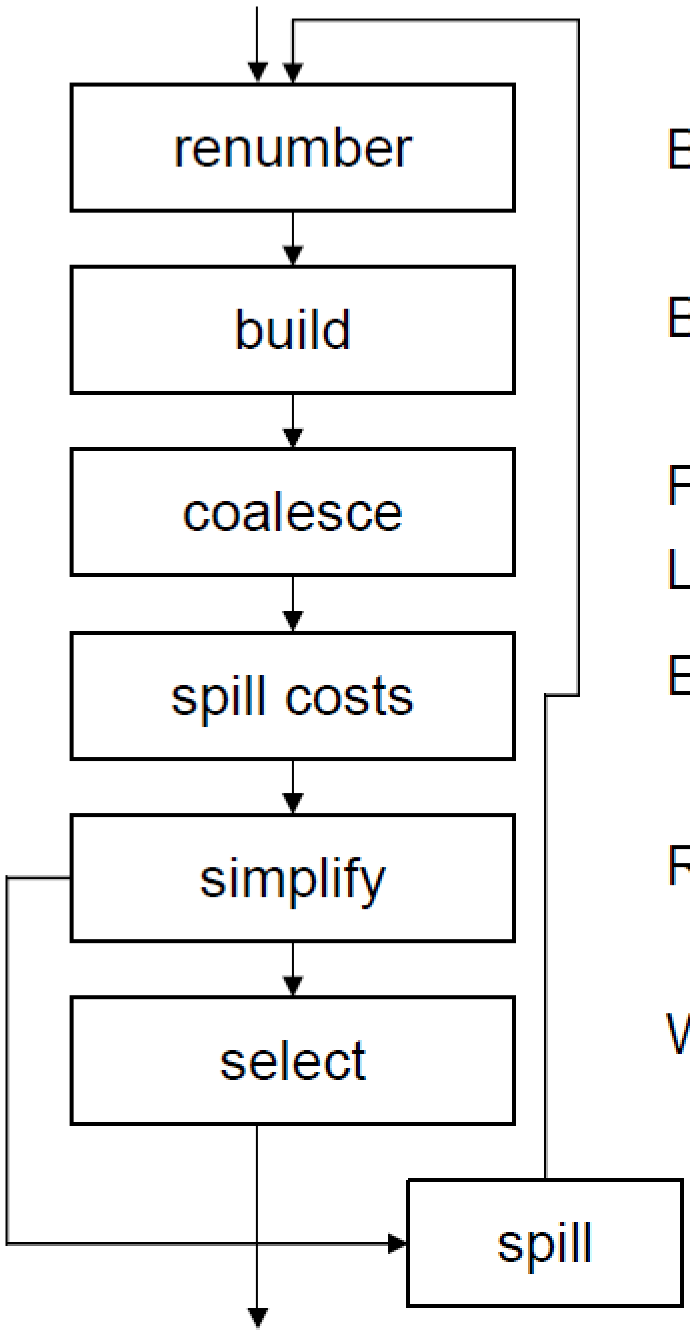
Example (3 Registers)



Example (3 Registers)



Chaitin Algorithm



Build SSA, build live ranges, rename

Build the interference graph

Fold unneeded copies

$LR_x \rightarrow LR_y$, and $<LR_x, LR_y> \notin G_I \Rightarrow$ combine LR_x & LR_y

Estimate cost for spilling
each live range

Remove nodes from the graph

while *N* is non-empty
if ∃ *n* with *n*° < k then
push *n* onto stack
else pick *n* to spill
push *n* onto stack
remove *n* from G_I

While stack is non-empty
pop *n*, insert *n* into G_I, & try to color it

Spill uncolored definitions & uses