Module 11: Compilers: Global Register Allocation

Profs. Pralay Mitra and Partha Pratim Das

pralay@cse.iitkgp.ac.in, ppd@cse.iitkgp.ac.in

October 29 & November 4, 2019

Department of Computer Science & Engineering

Outline

- Issues in Global Register Allocation
- The Problem
- Register Allocation based on Usage Counts
- Chaitin's graph coloring based algorithm

Some Issues in Register Allocation

- Which values in a program reside in registers? (register allocation)
- In which register? (register assignment)
  - The two together are usually loosely referred to as register allocation
- What is the unit at the level of which register allocation is done?
  - Typical units are basic blocks, functions and regions.
  - RA within basic blocks is called local RA
  - The other two are known as global RA
  - Global RA requires much more time than local RA

Some Issues in Register Allocation

- Phase ordering between register allocation and instruction scheduling
  - Performing RA first restricts movement of code during scheduling – not recommended
  - Scheduling instructions first cannot handle spill code introduced during RA
    - Requires another pass of scheduling
- Tradeoff between speed and quality of allocation
  - In some cases, e.g., in Just-In-Time compilation, cannot afford to spend too much time in register allocation
  - Only local or both local and global allocation?

The Problem

- Global Register Allocation assumes that allocation is done beyond basic blocks and usually at function level
- Decision problem related to register allocation :
  - Given an intermediate language program represented as a control flow graph and a number ***k***, is there an assignment of registers to program variables such that no conflicting variables are assigned the same register, no extra loads or stores are introduced, and at most ***k*** registers are used.
- This problem has been shown to be NP-hard (Sethi 1970).
- Graph colouring is the most popular heuristic used.
- However, there are simpler algorithms as well

Conflicting variables

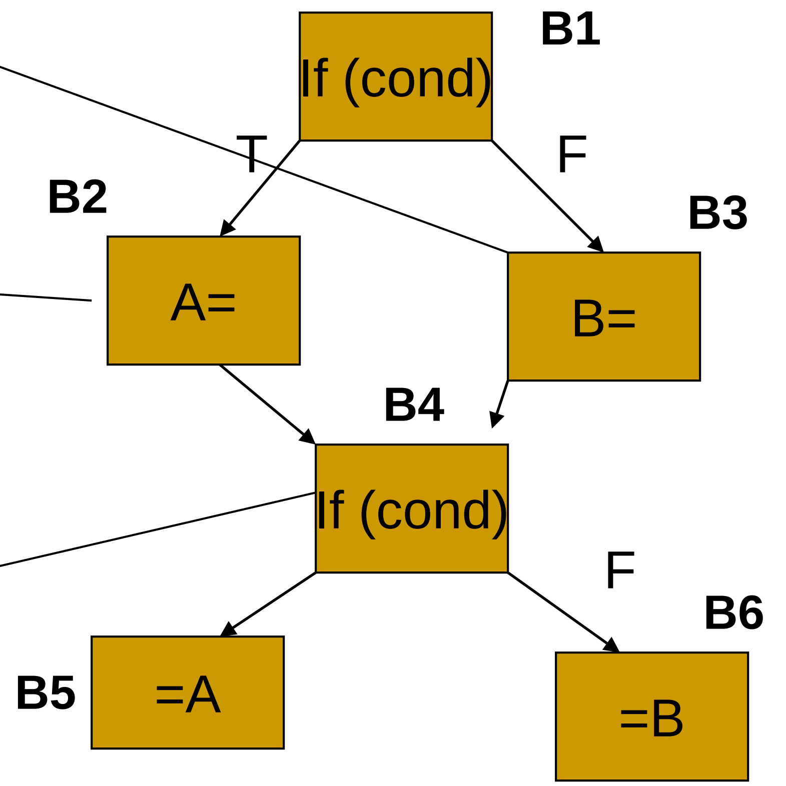
- Two variables interfere or conflict if their live ranges intersect
  - A variable is live at a point *p* in the flow graph, if there is a use of that variable in the path from *p* to the end of the flow graph
  - The live range of a variable is the smallest set of program points at which it is live.
  - Typically, instruction no. in the basic block along with the basic block no. is the representation for a point.

Example

Live range of A: B2, B4 B5
Live range of B: B3, B4, B6

If (cond) A not live
then A =
else B =
X: if (cond) B not live
then = A
else = B

A and B both live



Global Register Allocation via Usage Counts (for Single Loops)

- Allocate registers for variables used within loops
- Requires information about liveness of variables at the entry and exit of each basic block (BB) of a loop
- Once a variable is computed into a register, it stays in that register until the end of the BB (subject to existence of next-uses)
- Load/Store instructions cost 2 units (because they occupy two words)

Global Register Allocation via Usage Counts (for Single Loops)

1. For every usage of a variable ***v*** in a BB, until it is first defined, do:
   - savings(v) = savings(v) + 1
   - after v is defined, it stays in the register any way, and all further references are to that register
2. For every variable **v** computed in a BB, if it is live on exit from the BB,
   - count a savings of 2, since it is not necessary to store it at the end of the BB

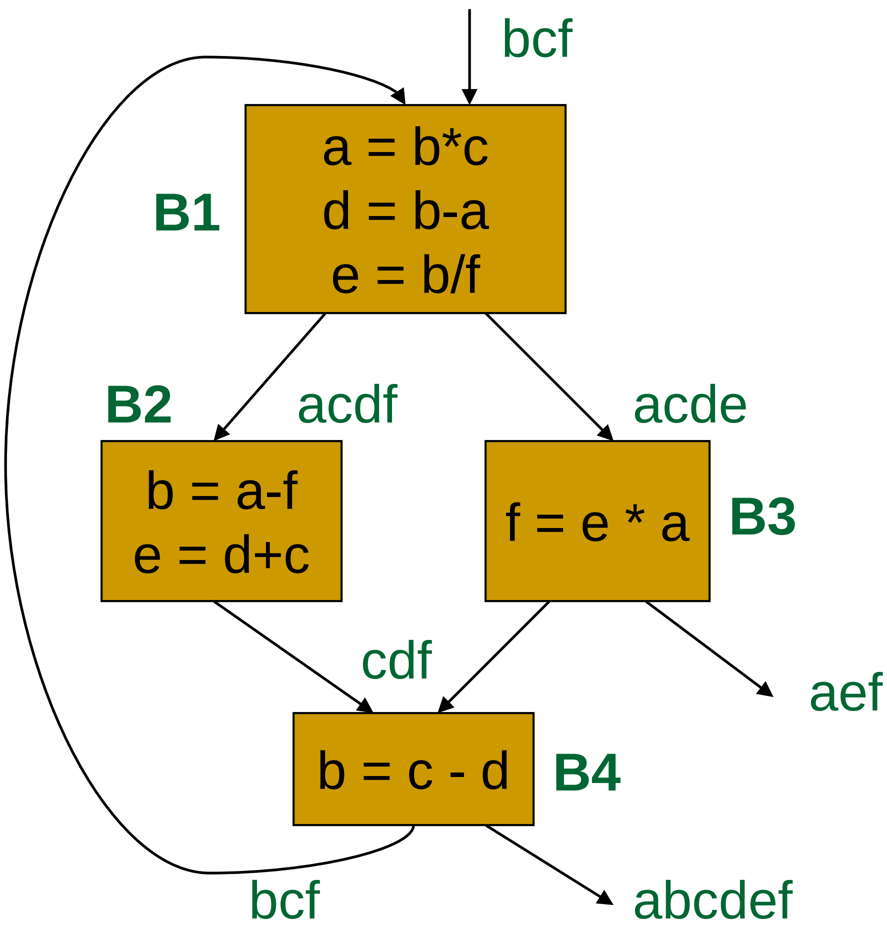
Global Register Allocation via Usage Counts (for Single Loops)

- Total savings per variable ***v*** are

$$\sum_{B \in Loop} (savings(v, B) + 2 * liveandcomputed(v, B))$$

  - *liveandcomputed(v,B)* in the second term is 1 or 0
- On entry to (exit from) the loop, we load (store) a variable live on entry (exit), and lose 2 units for each
  - But, these are "one time" costs and are neglected
- Variables, whose savings are the highest will reside in registers

Global Register Allocation via Usage Counts (for Single Loops)



Savings for the variables

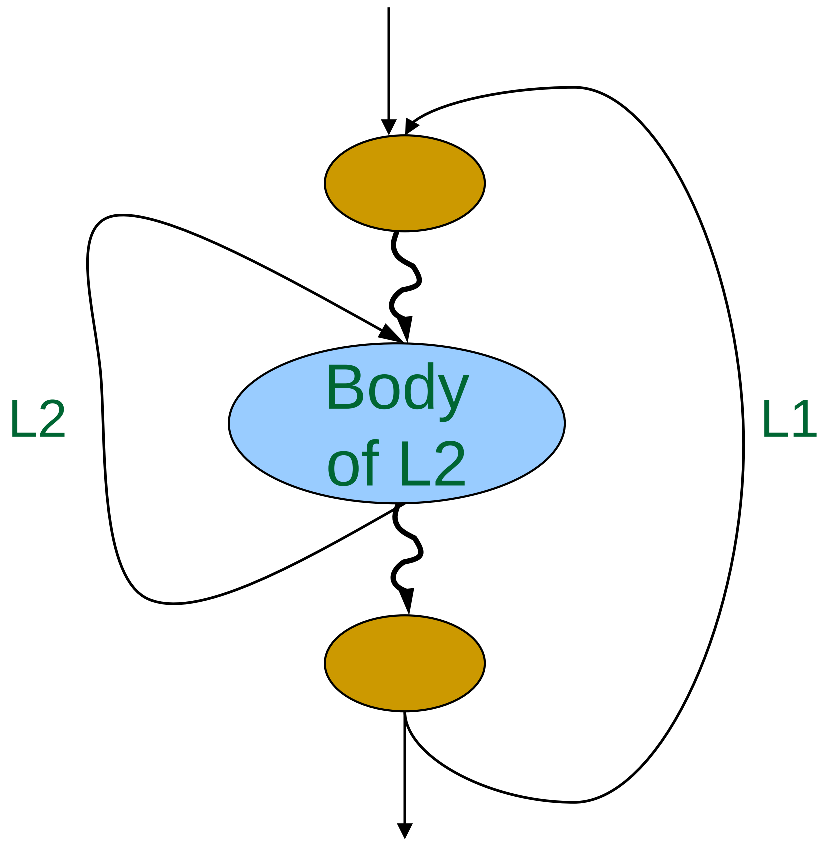
	B1	B2	B3	B4	
a:	(0+2)+	(1+0)+	(1+0)+	(0+0)	= 4
b:	(3+0)+	(0+0)+	(0+0)+	(0+2)	= 5
c:	(1+0)+	(1+0)+	(0+0)+	(1+0)	= 3
d:	(0+2)+	(1+0)+	(0+0)+	(1+0)	= 4
e:	(0+2)+	(0+0)+	(1+0)+	(0+0)	= 3
f:	(1+0)+	(1+0)+	(0+2)+	(0+0)	= 4

If there are 3 registers, they will be allocated to the variables, a, b, and d

Global Register Allocation via Usage Counts (for Nested Loops)

- We first assign registers for inner loops and then consider outer loops. Let L1 nest L2
- For variables assigned registers in L2, but not in L1
  - load these variables on entry to L2 and store them on exit from L2
- For variables assigned registers in L1, but not in L2
  - store these variables on entry to L2 and load them on exit from L2
- All costs are calculated keeping the above rules

Global Register Allocation via Usage Counts (for Nested Loops)



- case 1: variables x,y,z assigned registers in L2, but not in L1
  - Load x,y,z on entry to L2
  - Store x,y,z on exit from L2
- case 2: variables a,b,c assigned registers in L1, but not in L2
  - Store a,b,c on entry to L2
  - Load a,b,c on exit from L2
- case 3: variables p,q assigned registers in both L1 and L2
  - No special action

Chaitin’s Formulation of the Register Allocation Problem

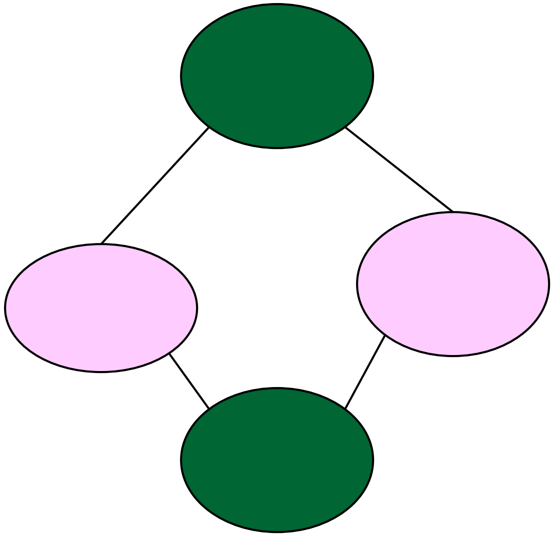
- A graph colouring formulation on the interference graph
- Nodes in the graph represent either live ranges of variables or entities called webs
- An edge connects two live ranges that interfere or conflict with one another
- Usually both adjacency matrix and adjacency lists are used to represent the graph.

Chaitin's Formulation of the Register Allocation Problem

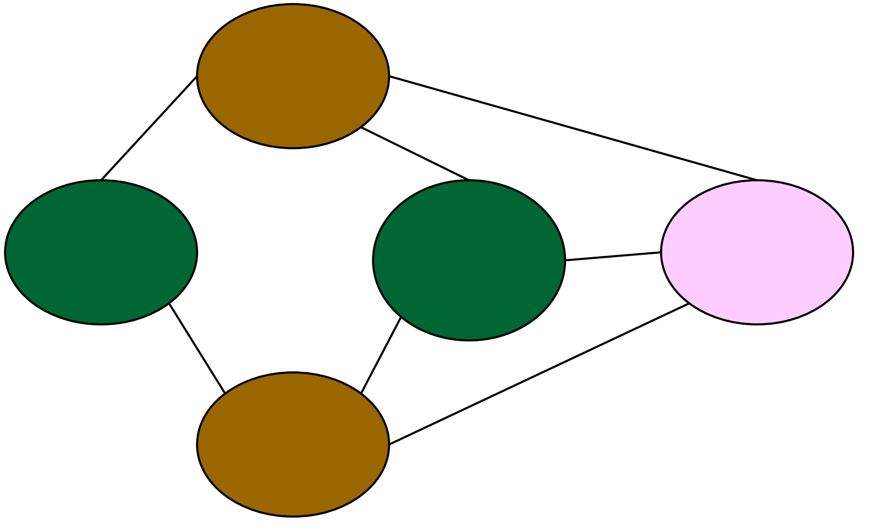
- Assign colours to the nodes such that two nodes connected by an edge are not assigned the same colour
  - The number of colours available is the number of registers available on the machine
  - A k-colouring of the interference graph is mapped onto an allocation with k registers

Example

- Two colourable



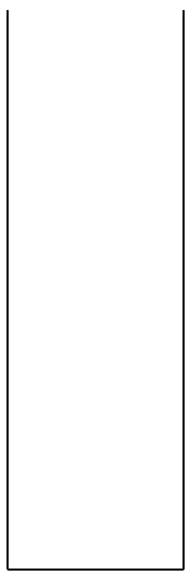
Three colourable



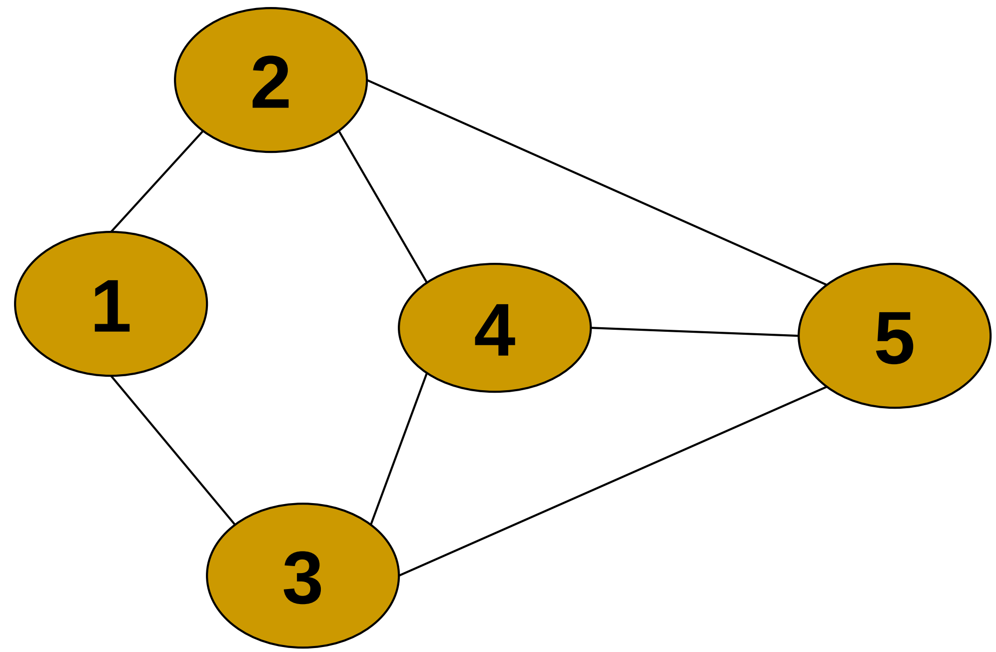
Idea behind Chaitin's Algorithm

- Choose an arbitrary node of degree less than k and put it on the stack
- Remove that vertex and all its edges from the graph
  - This may decrease the degree of some other nodes and cause some more nodes to have degree less than k
- At some point, if all vertices have degree greater than or equal to k, some node has to be spilled
- If no vertex needs to be spilled, successively pop vertices off stack and colour them in a colour not used by neighbours (reuse colours as far as possible)

Simple example – Given Graph

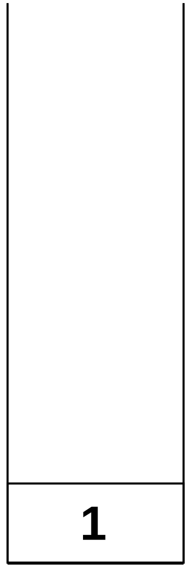


STACK

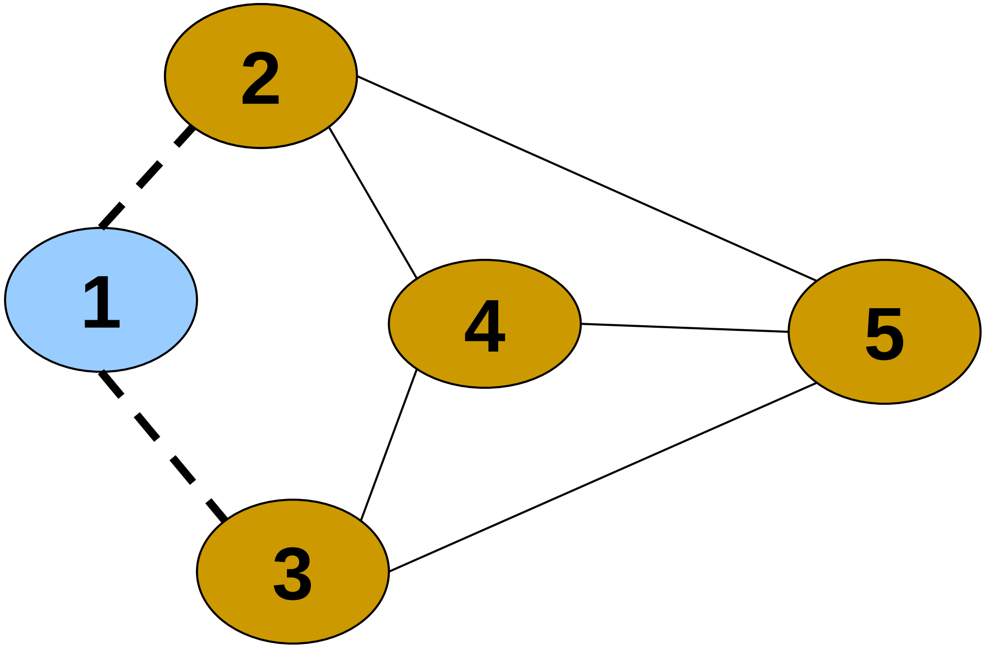


3 REGISTERS

Simple Example – Delete Node 1

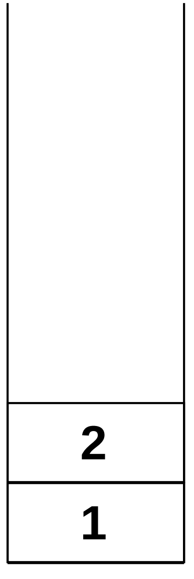


STACK

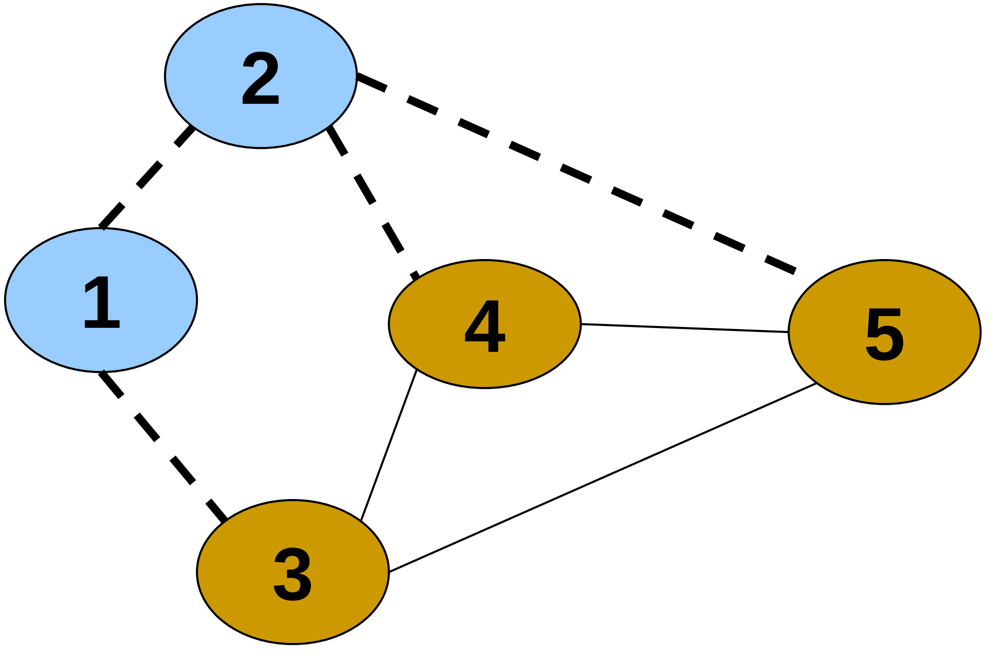


3 REGISTERS

Simple Example – Delete Node 2

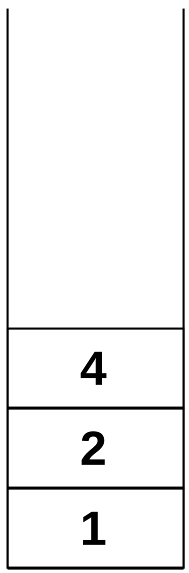


STACK

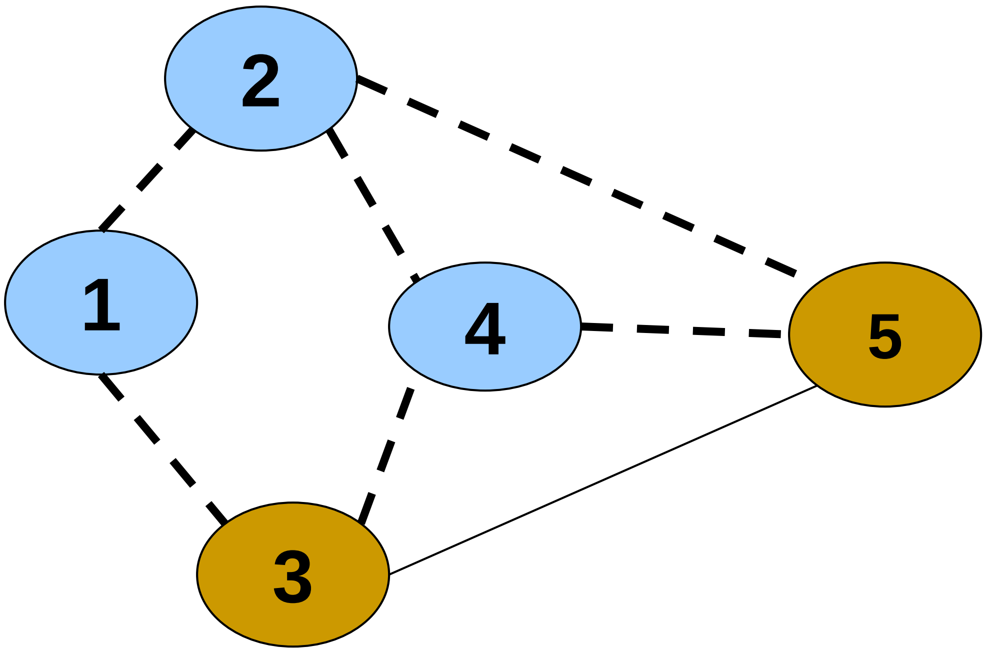


3 REGISTERS

Simple Example – Delete Node 4

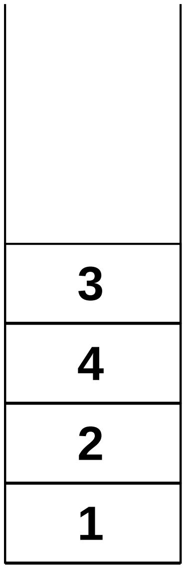


STACK

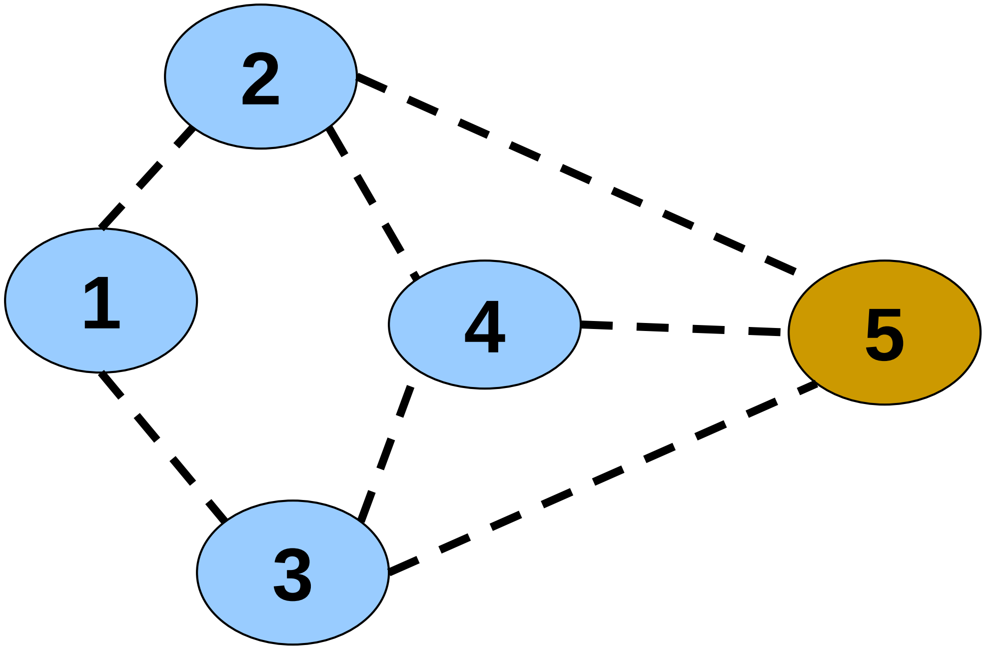


3 REGISTERS

Simple Example – Delete Nodes 3

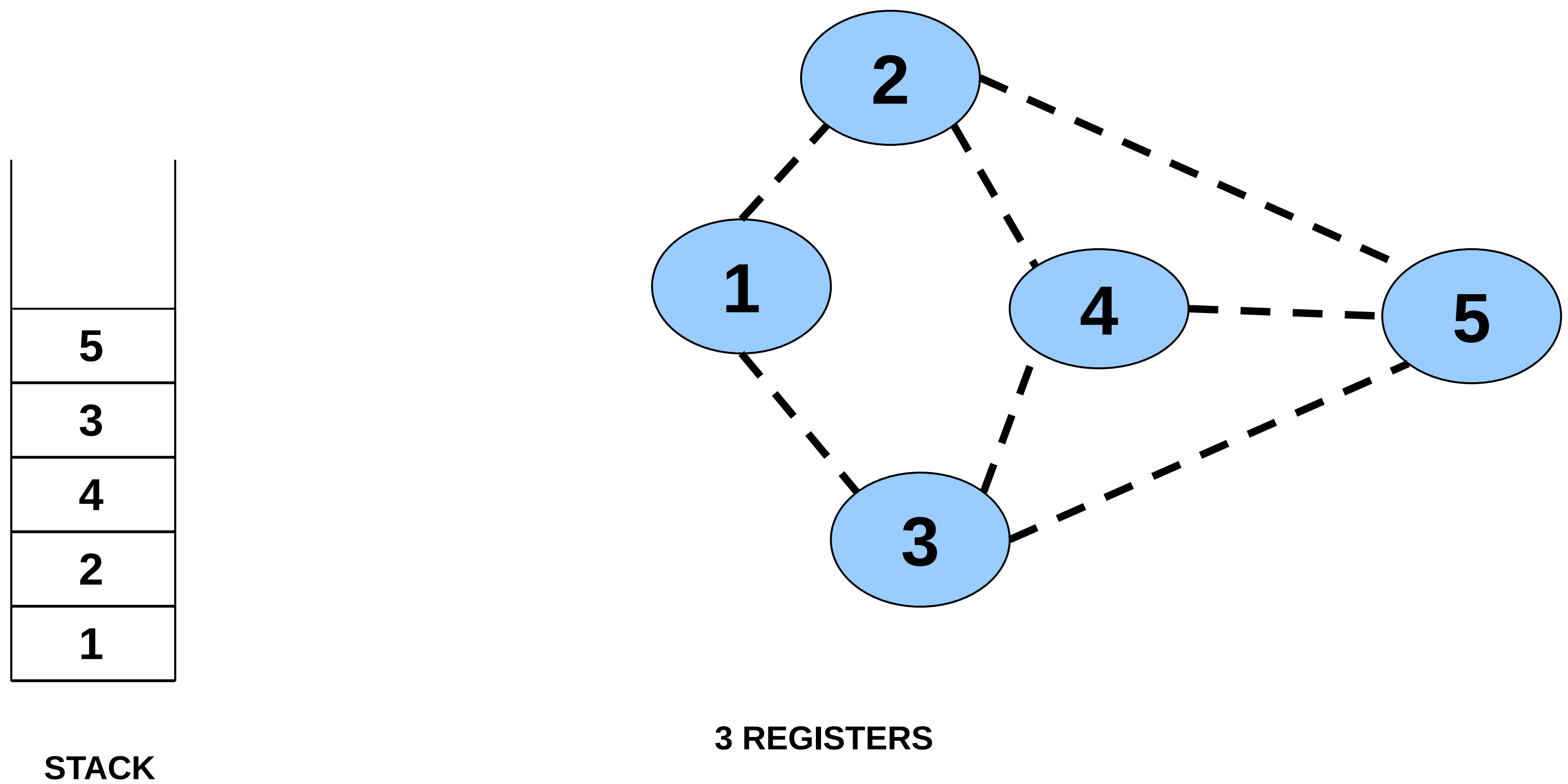


STACK

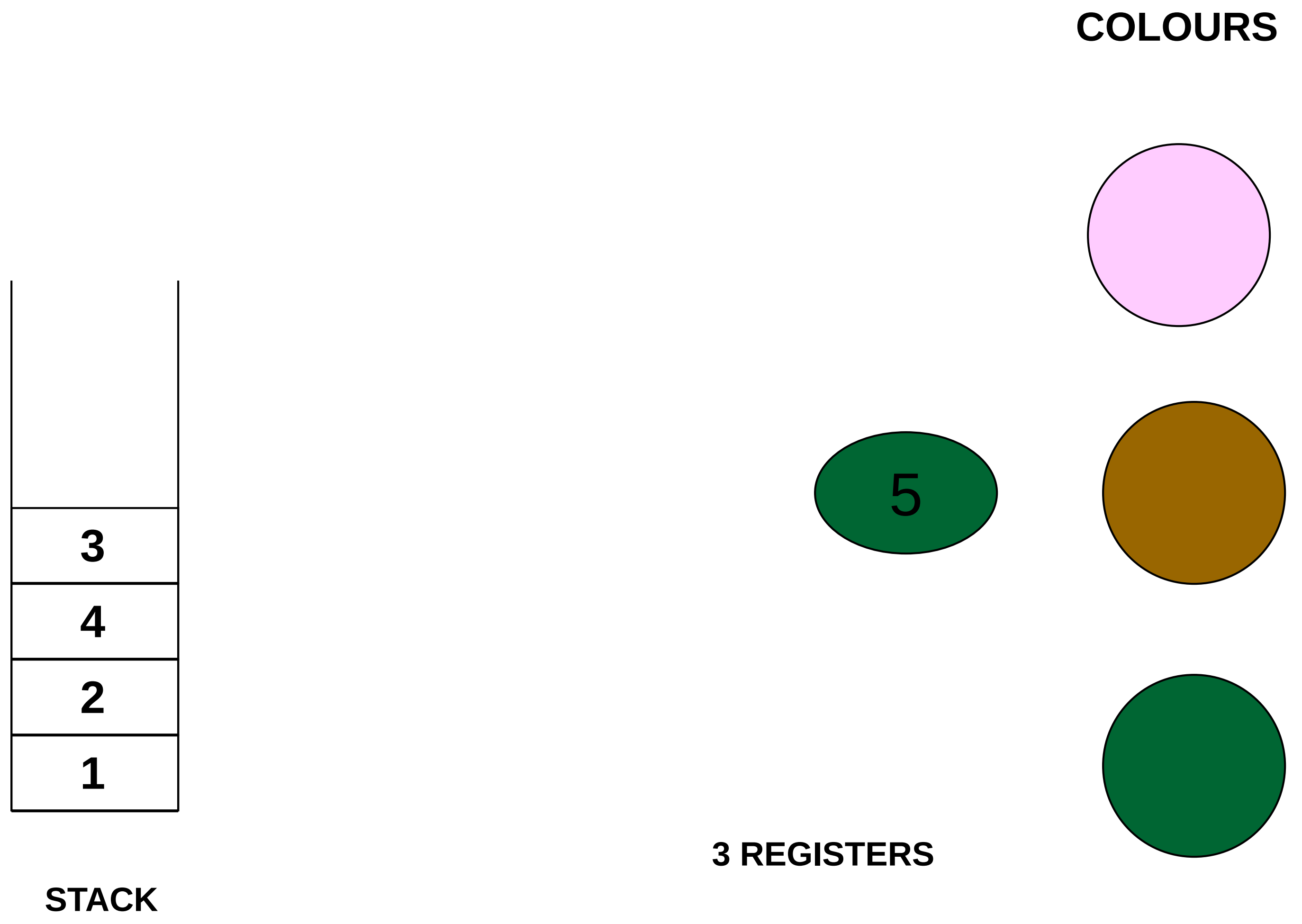


3 REGISTERS

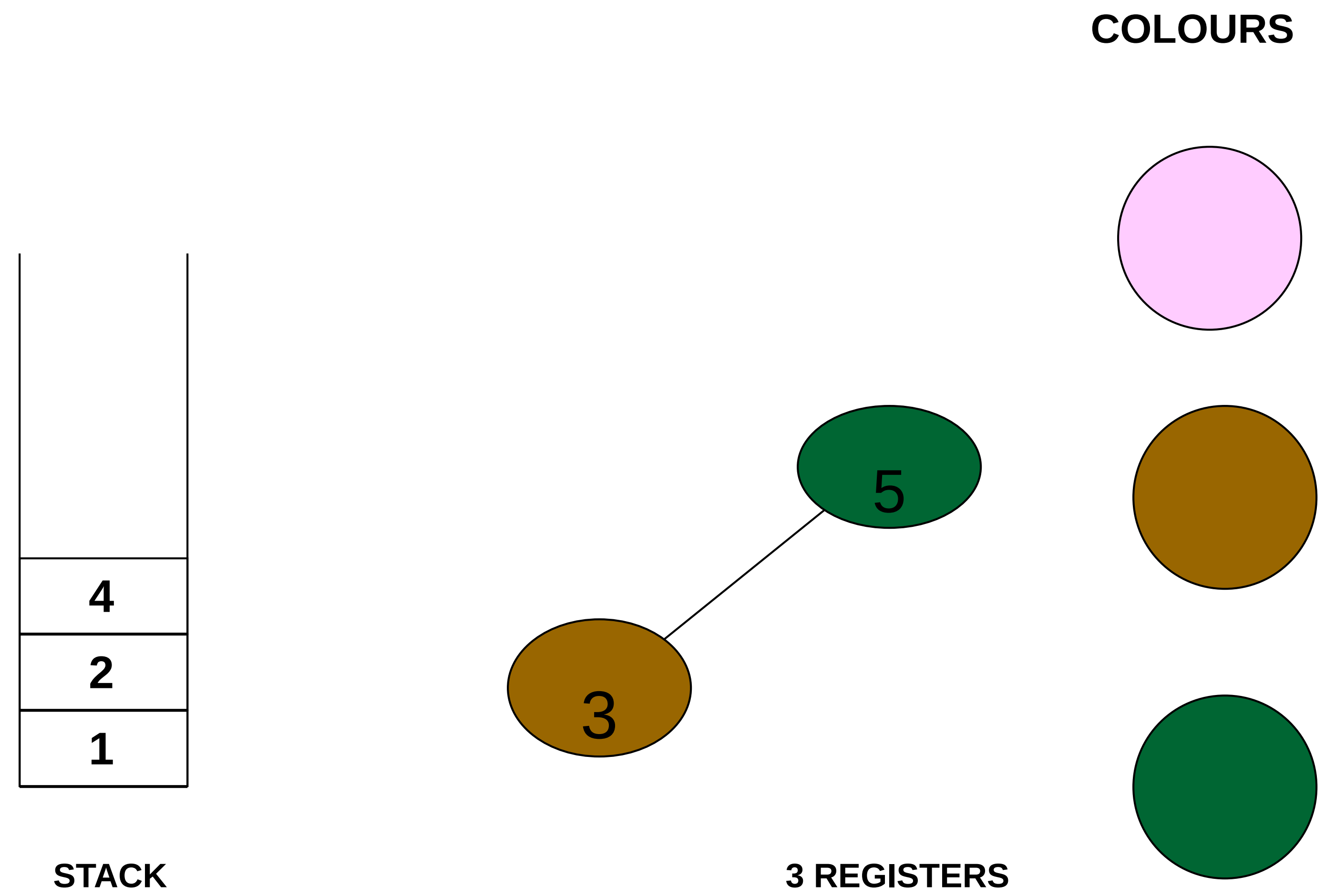
Simple Example – Delete Nodes 5



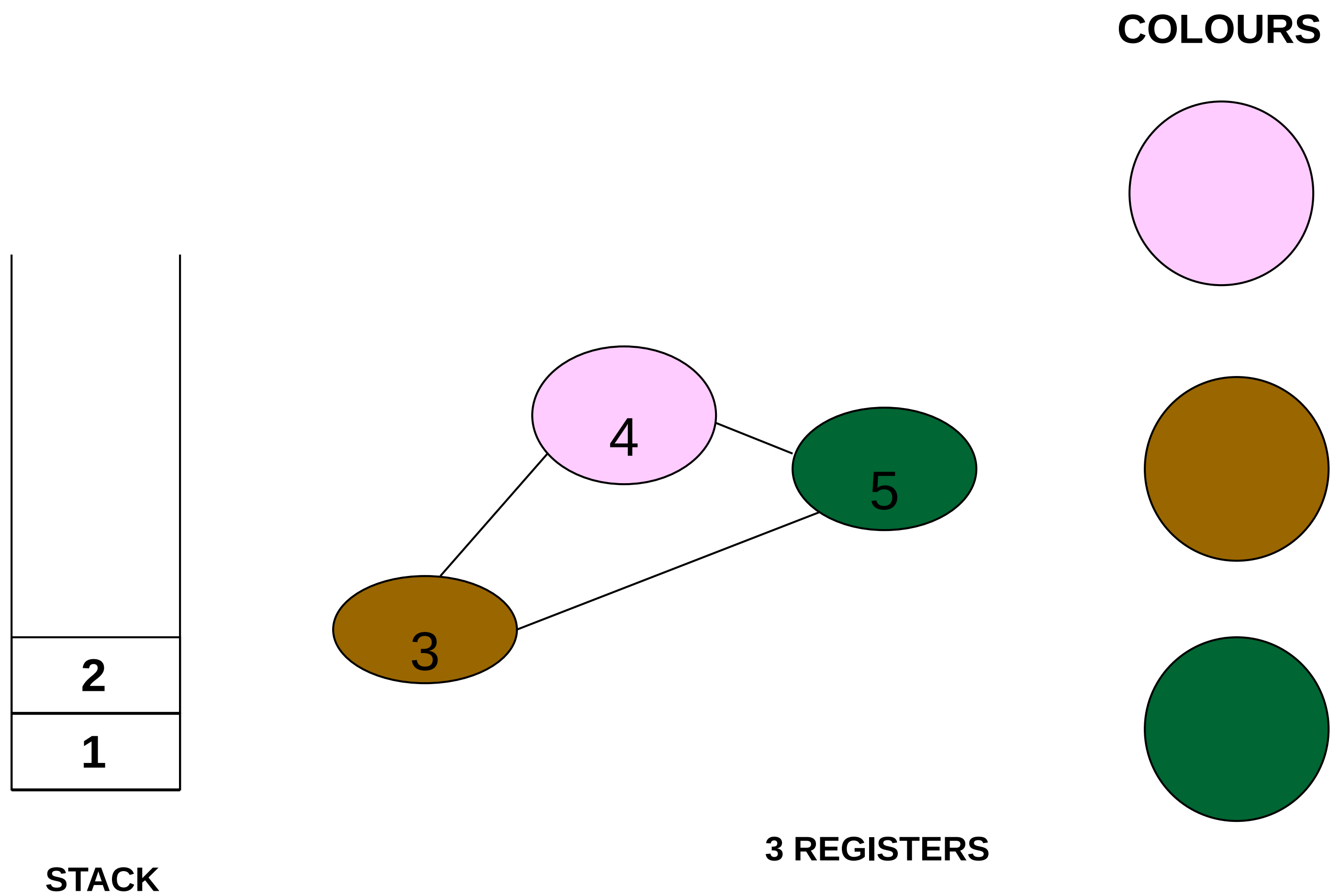
Simple Example – Colour Node 5



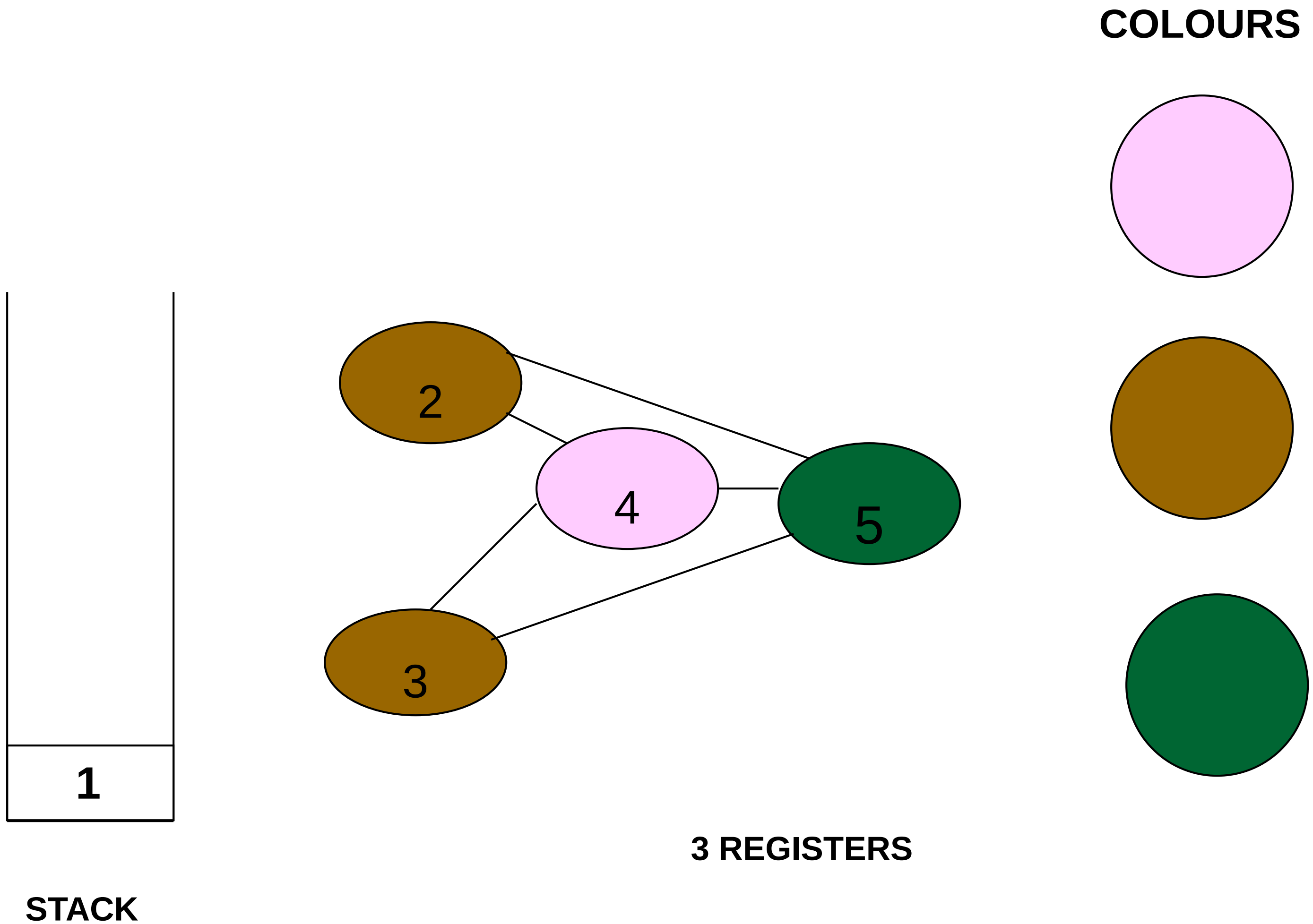
Simple Example – Colour Node 3



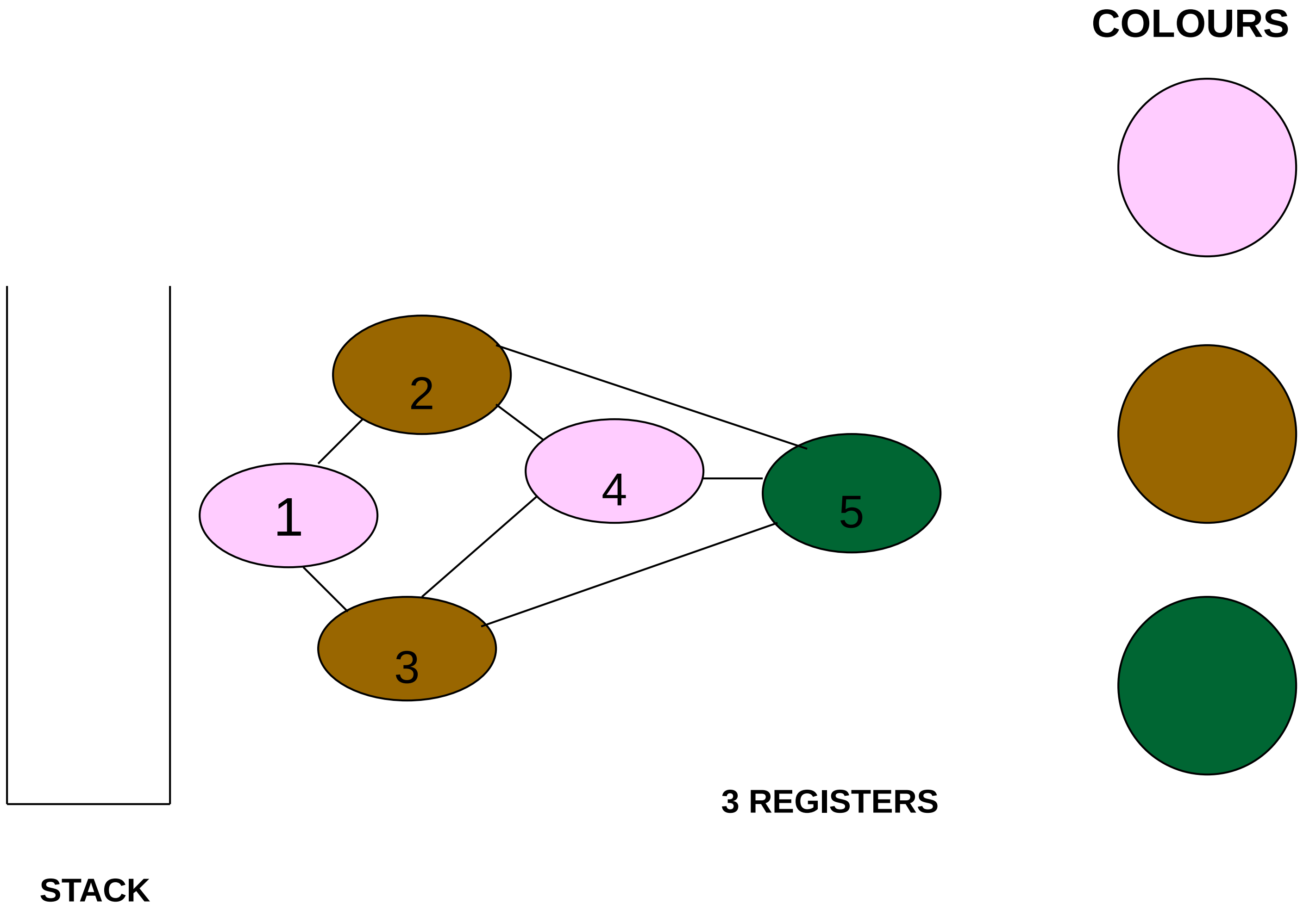
Simple Example – Colour Node 4



Simple Example – Colour Node 2



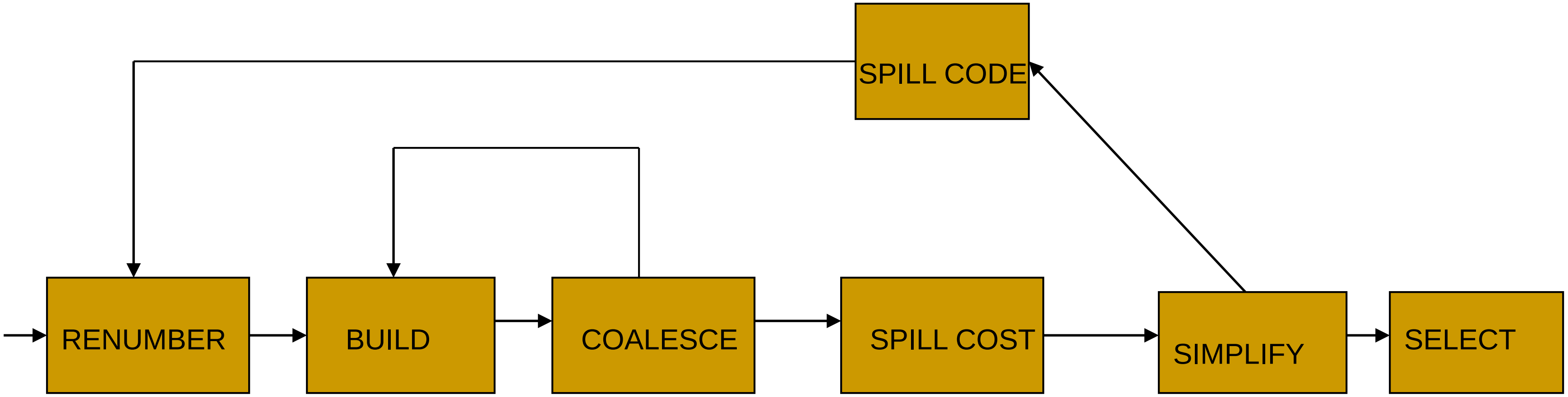
Simple Example – Colour Node 1



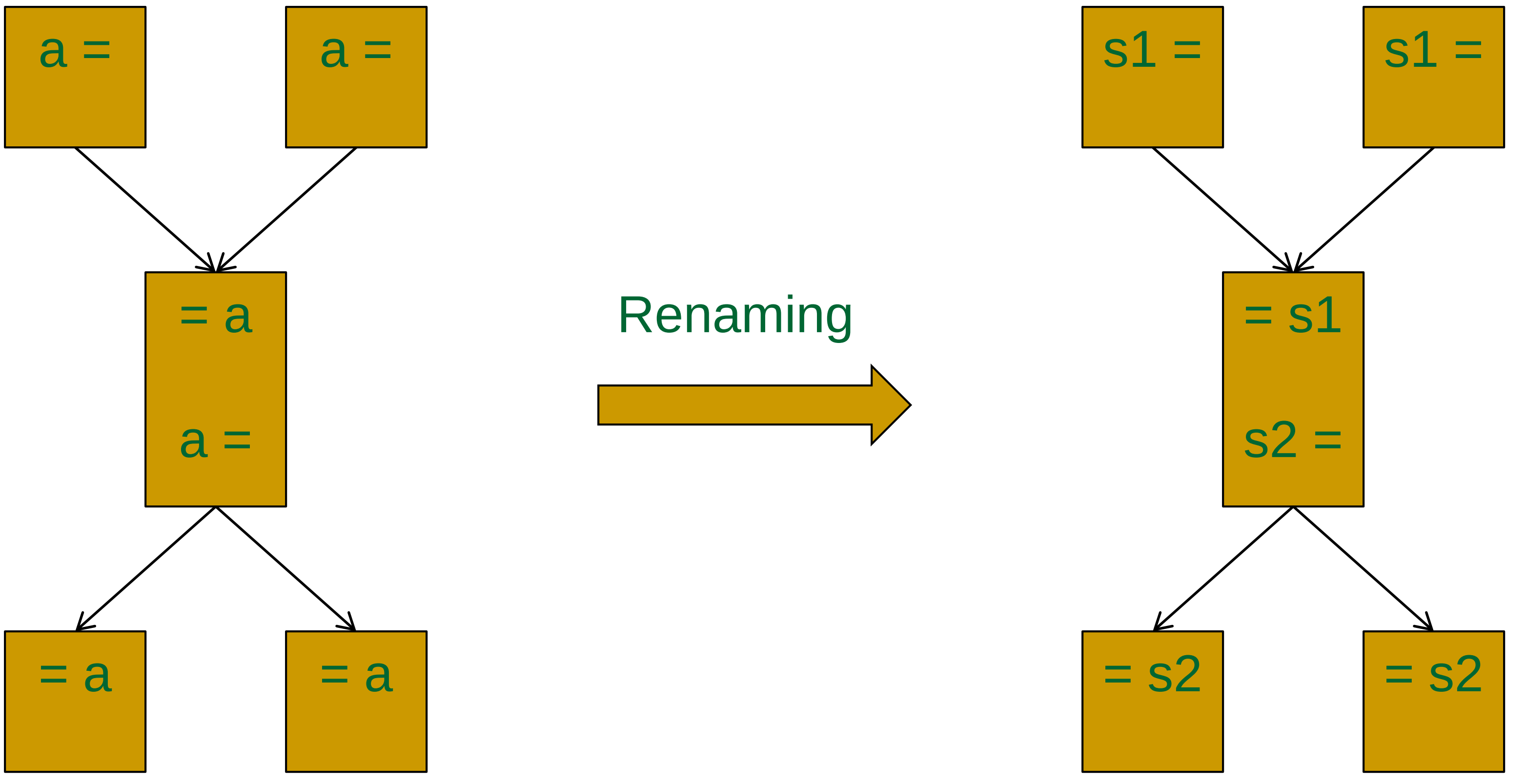
Steps in Chaitin's Algorithm

- Identify units for allocation
  - Renames variables/symbolic registers in the IR such that each live range has a unique name (number)
- Build the interference graph
- Coalesce by removing unnecessary move or copy instructions
- Colour the graph, thereby selecting registers
- Compute spill costs, simplify and add spill code till graph is colourable

The Chaitin Framework



Example of Renaming



An Example

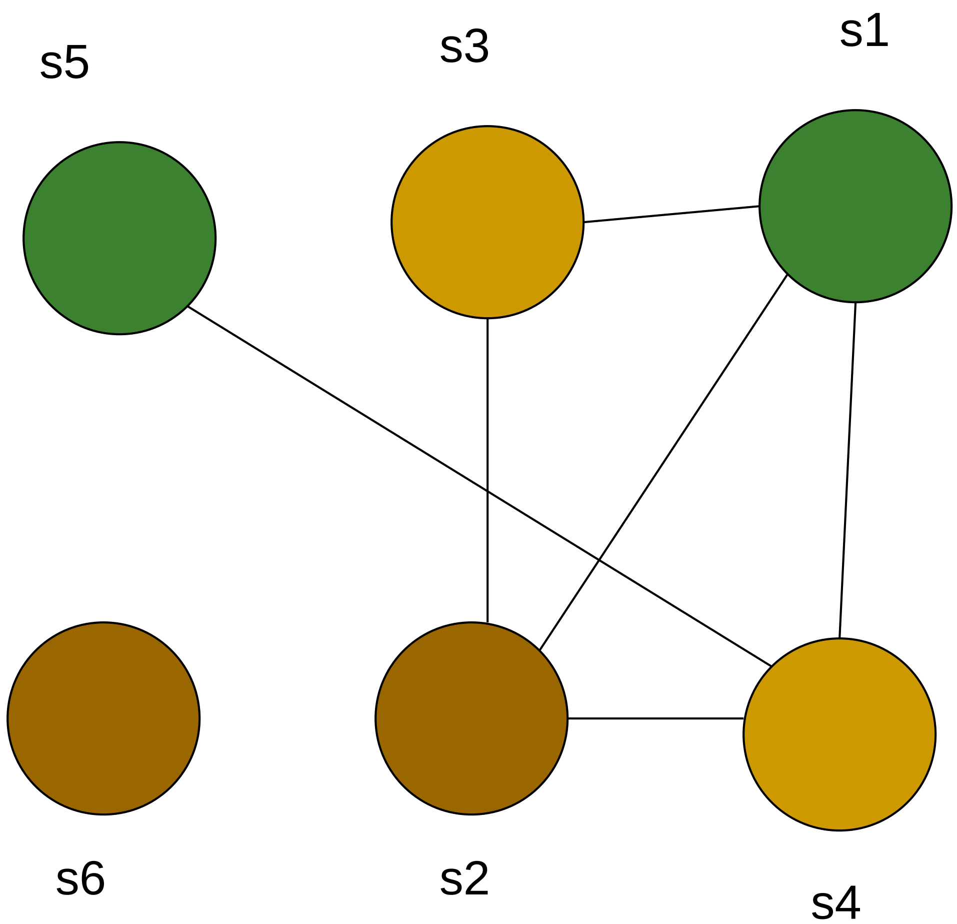
Original code

1. x= 2
2. y = 4
3. w = x+ y
4. z = x+1
5. u = x*y
6. x= z*2

Code with symbolic registers

1. s1=2; (lv of s1: 1-5)
2. s2=4; (lv of s2: 2-5)
3. s3=s1+s2; (lv of s3: 3-3)
4. s4=s1+1; (lv of s4: 4-6)
5. s5=s1*s2; (lv of s5: 5-5)
6. s6=s4*2; (lv of s6: 6- ...)

Stack Order for Colouring & Register Allocation



INTERFERENCE GRAPH

s1 → r1
s2 → r2
s3 → r3
s4 → r3
s5 → r1
s6 → r2

1. x= 2
2. y = 4
3. w = x+ y
4. z = x+1
5. u = x*y
6. x= z*2

1. s1=2; (lv of s1: 1-5)
2. s2=4; (lv of s2: 2-5)
3. s3=s1+s2; (lv of s3: 3-3)
4. s4=s1+1; (lv of s4: 4-6)
5. s5=s1*s2; (lv of s5: 5-5)
6. s6=s4*2; (lv of s6: 6- ...)

Example(continued)

1. x= 2
2. y = 4
3. w = x+ y
4. z = x+1
5. u = x*y
6. x= z*2

s1 → r1
s2 → r2
s3 → r3
s4 → r3
s5 → r1
s6 → r2

1. s1=2; (lv of s1: 1-5)
2. s2=4; (lv of s2: 2-5)
3. s3=s1+s2; (lv of s3: 3-3)
4. s4=s1+1; (lv of s4: 4-6)
5. s5=s1*s2; (lv of s5: 5-5)
6. s6=s4*2; (lv of s6: 6- ...)

Final code: 3 registers are sufficient for no spills

r1= 2

r2= 4

r3= r1+r2

r3= r1+1

r1= r1*r2

r2= r3*2