Module 10

Das

Module 10: CS31003: Compilers

Global Register Allocation

Partha Pratim Das

Department of Computer Science
Ashoka University

ppd@ashoka.edu.in, partha.das@ashoka.edu.in, 9830030880

December 07, 2022

Module Objectives

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

Exclude slide 10.09-10.14

Module Outline

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 (RA)**
- 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
  - RA within *functions* and *regions* are known as global RA
  - Global RA requires lot more time than local RA

Some Issues in Register Allocation

- Phase ordering between *register allocation* and *instruction scheduling*
- In which register? (*register assignment*)
  - 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, for example, 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
  - The representation for a point is:
    - basic block number
    - instruction number in the basic block

Module 10

Das

Objectives & Outline

Issues in Register Allocation

The Problem

GRA by Usage Count

Bubble Sort

Chaitin's Algorithm: GRA by Graph Coloring

Graph Coloring

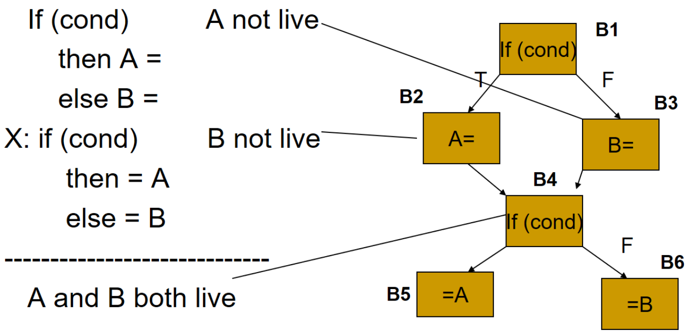
Framework

Example

Register Spill

Example

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



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

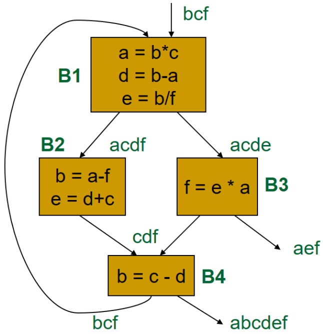
Global Register Allocation via Usage Counts (for Single Loops)

- 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

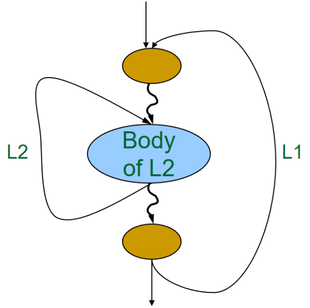
	B1	B2	B3	B4	
a:	**(0+2)**+	**(1+0)**+	**(1+0)**+	**(0+0)**=	**4**
b:	**(3+0)**+	**(0+0)**+	**(0+0)**+	**(0+2)**=	**5**
c:	(1+0)+	(1+0)+	(0+0)+	(1+0)=	3
d:	**(0+2)**+	**(1+0)**+	**(0+0)**+	**(1+0)**=	**4**
e:	(0+2)+	(0+0)+	(1+0)+	(0+0)=	3
f:	(1+0)+	(1+0)+	(0+2)+	(0+0)=	4

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

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

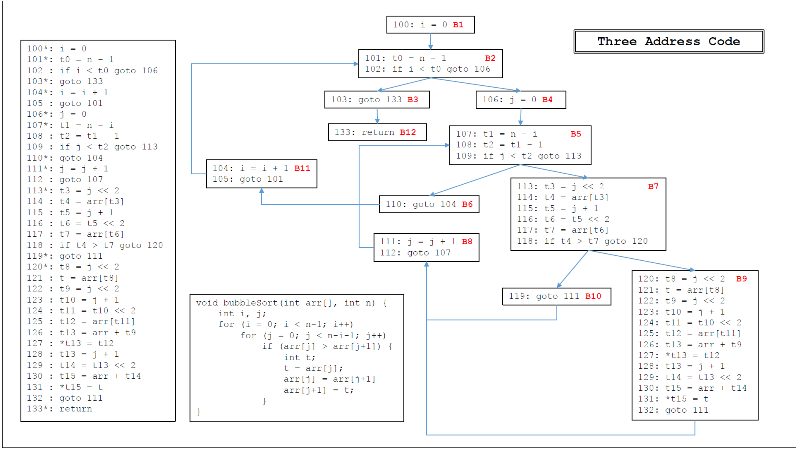
Global Register Allocation via Usage Counts (for Nested Loops)



- **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

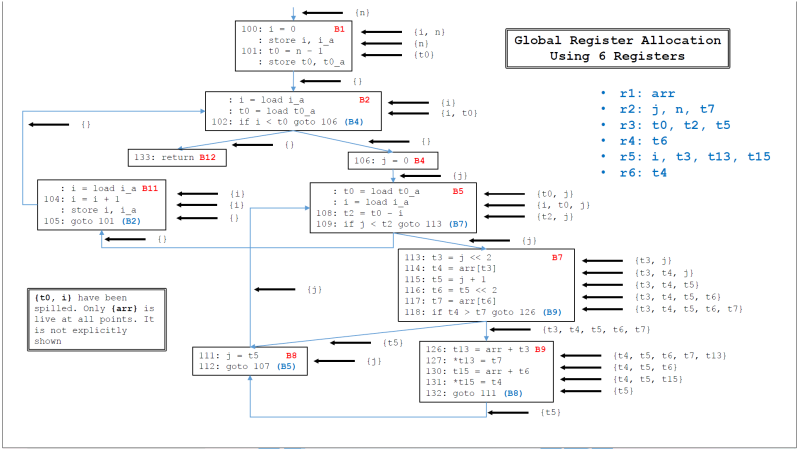
Sample Code Generation: Bubble Sort Three Address Code

- Three Address Code for Bubble Sort as generated by syntax directed translation



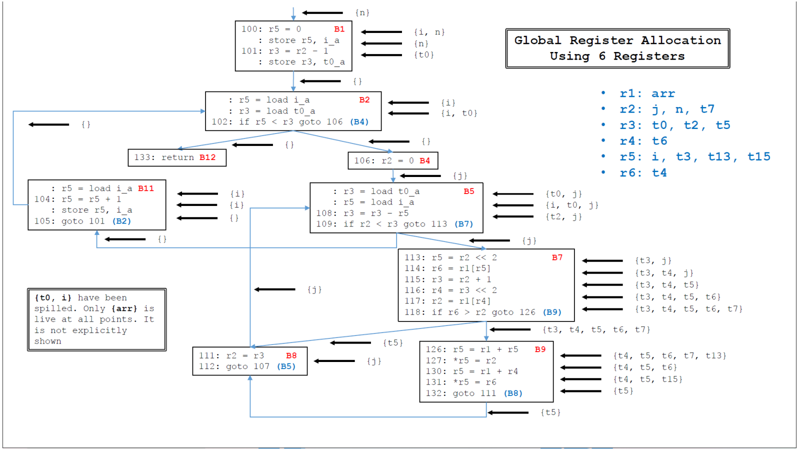
Sample Code Generation: Bubble Sort Liveness after LCSE, GCSE Optimization

- Three Address Code Optimized by peephole, LCSE by VN and GCSE by DFA. Finally, live variables are computed by DFA



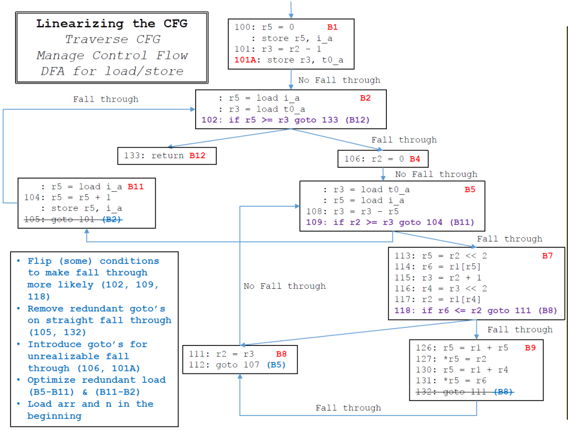
Sample Code Generation: Bubble Sort
Global Register Allocation

- Registers are allocated globally using graph coloring based on the liveness information
- Variables are replaced by respective registers



Sample Code Generation: Bubble Sort
Linearized and Optimized Target Code

- The CFG is linearized and further optimized to get the final target code



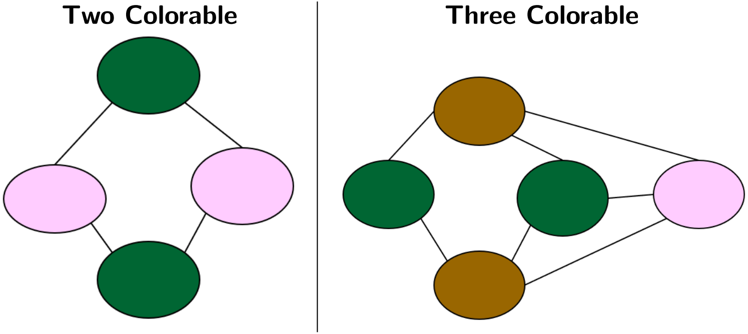

```
Final Code. # Reg. = 6

    : r1 = load arr
    : r2 = load n
B1  : r5 = 0
    : store r5, i_a
    : r3 = r2 - 1
    : store r3, t0_a
    : goto B2 // r5 has i
B5  : r3 = load t0_a
    : r5 = load i_a
    : r3 = r3 - r5
    : if r2 >= r3 goto B11 // r5 has i
B7  : r5 = r2 << 2
    : r6 = r1[r5]
    : r3 = r2 + 1
    : r4 = r3 << 2
    : r2 = r1[r4]
    : if r6 <= r2 goto B8
B9  : r5 = r1 + r5
    : *r5 = r2
    : r5 = r1 + r4
    : *r5 = r6
B8  : r2 = r3
    : goto B5
B11: r5 = load i_a // load-load (B5)
B11: r5 = r5 + 1
    : store r5, i_a
B2  : r5 = load i_a // store-load (B11)
B2  : r3 = load t0_a
    : if r5 >= r3 goto B12
B4  : r2 = 0
    : goto B5
B12: return
```

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.
- 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

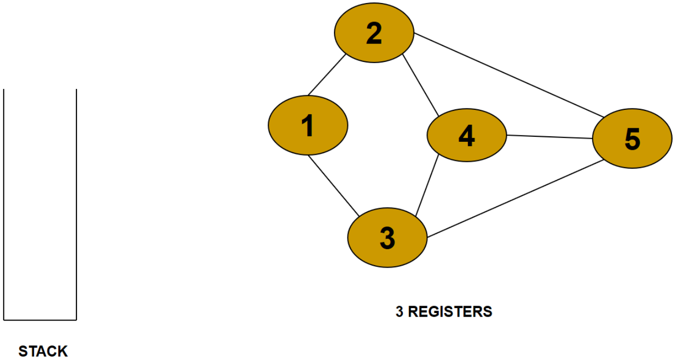
Example



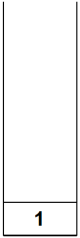
Idea behind Chaitin's Algorithm

- 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)

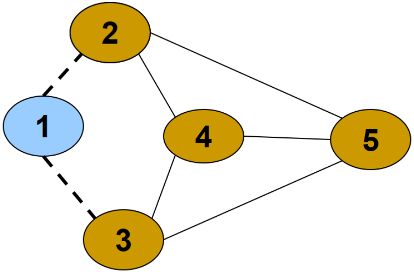
Simple example – Given Graph



Simple example – Delete Node 1

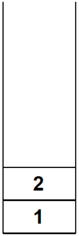


STACK

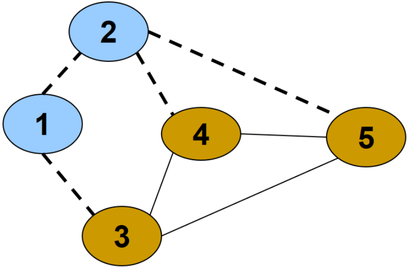


3 REGISTERS

Simple example – Delete Node 2

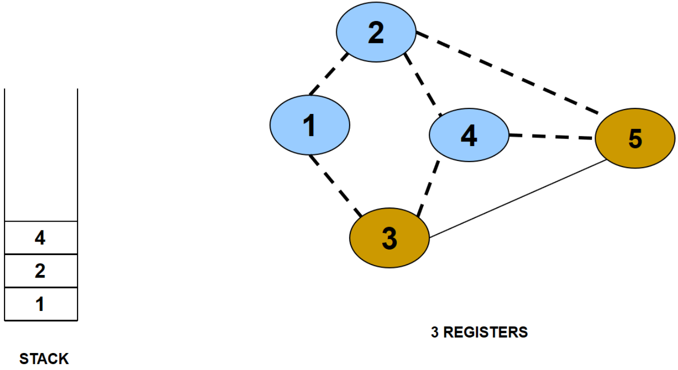


STACK

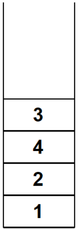


3 REGISTERS

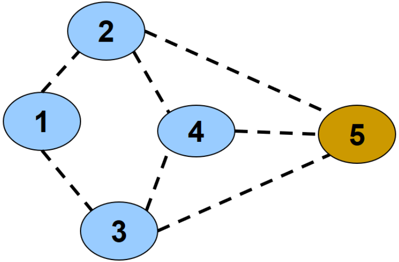
Simple example – Delete Node 4



Simple example – Delete Node 3

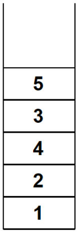


STACK

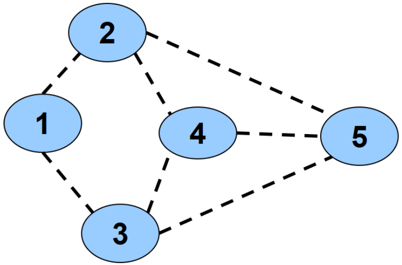


3 REGISTERS

Simple example – Delete Node 5

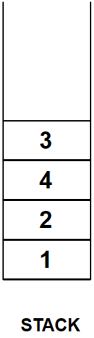


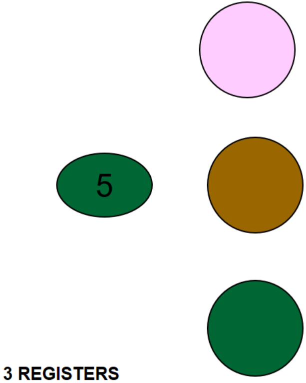
STACK



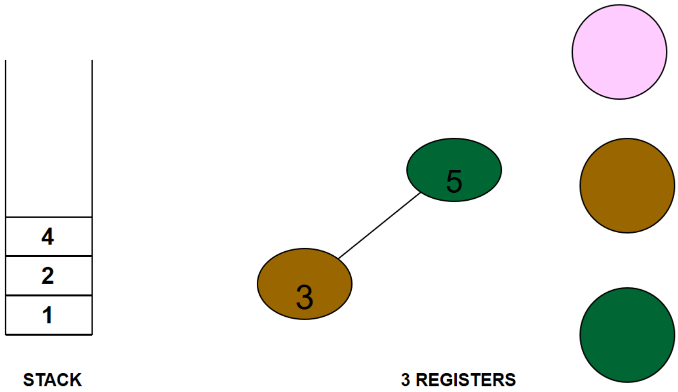
3 REGISTERS

Simple example – Colour Node 5

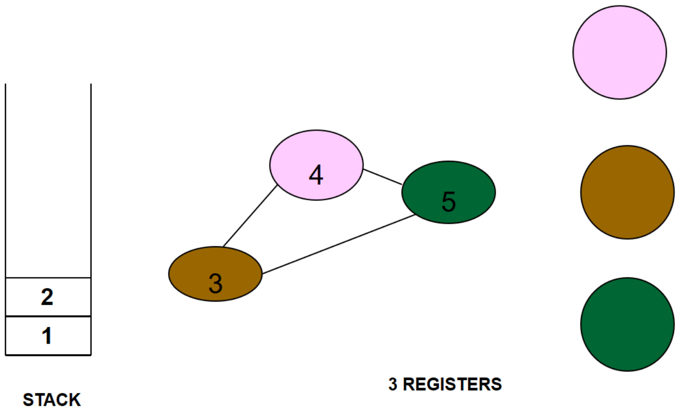




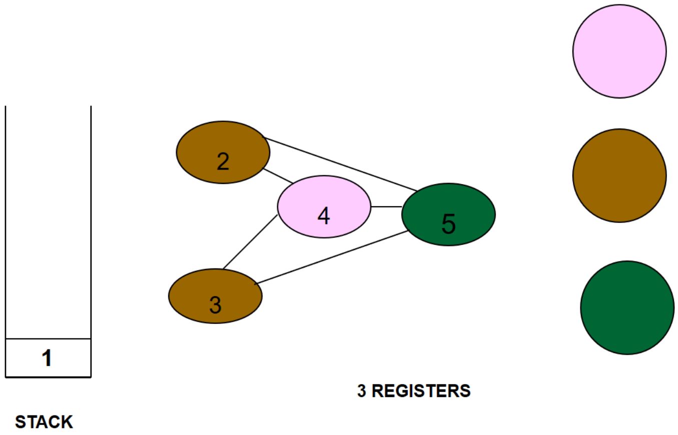
Simple example – Colour Node 3



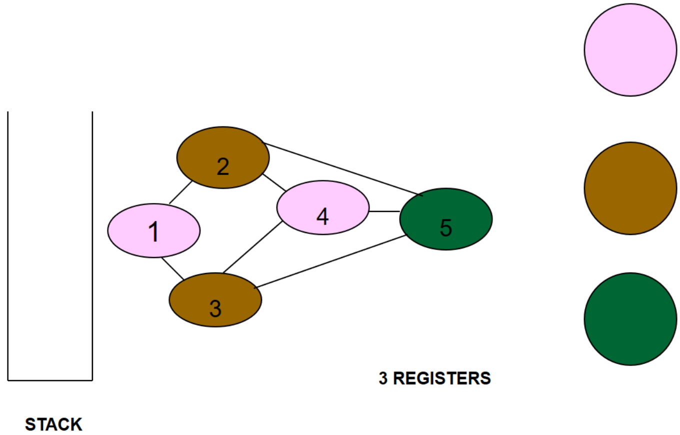
Simple example – Colour Node 4



Simple example – Colour Node 2



Simple example – Colour Node 1



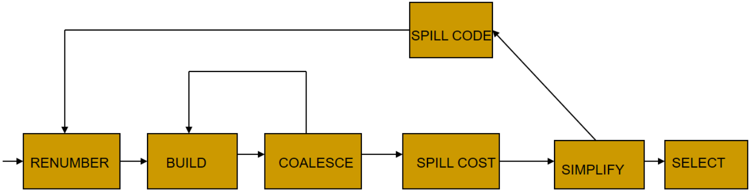
Steps in Chaitin's Algorithm

- 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

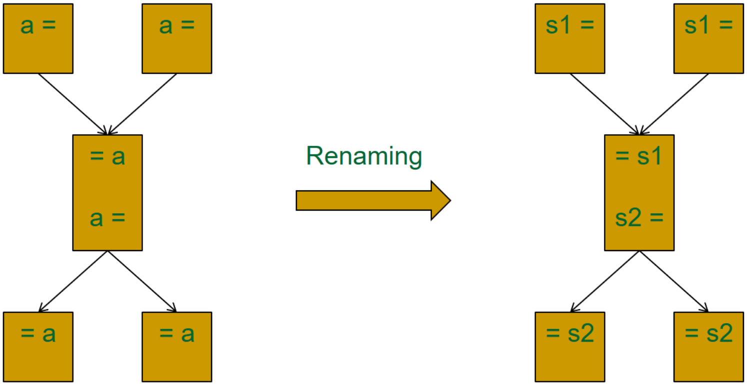
The Chaitin Framework

Module 10

Das



Example of Renaming

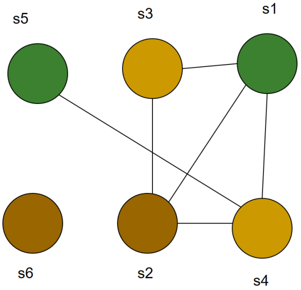


An Example

Original code	Code with symbolic registers	
1. x = 2	1. s1 = 2;	(lv of s1: 1-5)
2. y = 4	2. s2 = 4;	(lv of s2: 2-5)
3. w = x + y	3. s3 = s1 + s2;	(lv of s3: 3-3)
4. z = x + 1	4. s4 = s1 + 1;	(lv of s4: 4-6)
5. u = x * y	5. s5 = s1 * s2;	(lv of s5: 5-5)
6. x = z * 2	6. s6 = s4 * 2;	(lv of s6: 6-...)

An Example: Interference Graph

Interference Graph



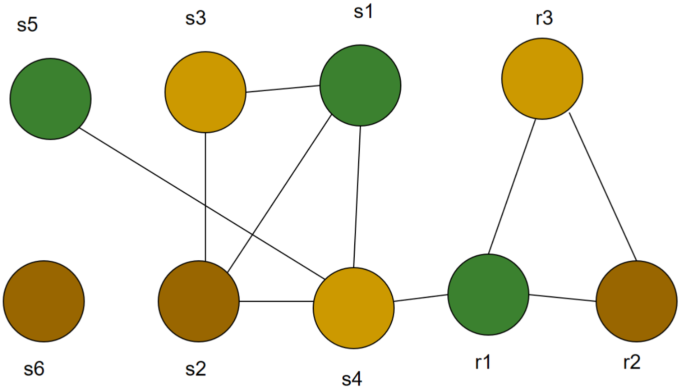
Stack Order for Colouring & Register Allocation (Number of Registers = 3)

s1	$\rightarrow$	r1
s2	$\rightarrow$	r2
s3	$\rightarrow$	r3
s4	$\rightarrow$	r3
s5	$\rightarrow$	r1
s6	$\rightarrow$	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- ...)

An Example: Interference Graph



Interference Graph

Here assume variable Z (s4) cannot occupy r1

An Example: Interference Graph

s1 → r1	1.	x = 2	
s2 → r2	2.	y = 4	
s3 → r3	3.	w = x + y	
s4 → r3	4.	z = x + 1	
s5 → r1	5.	u = x * y	
s6 → r2	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- ...)

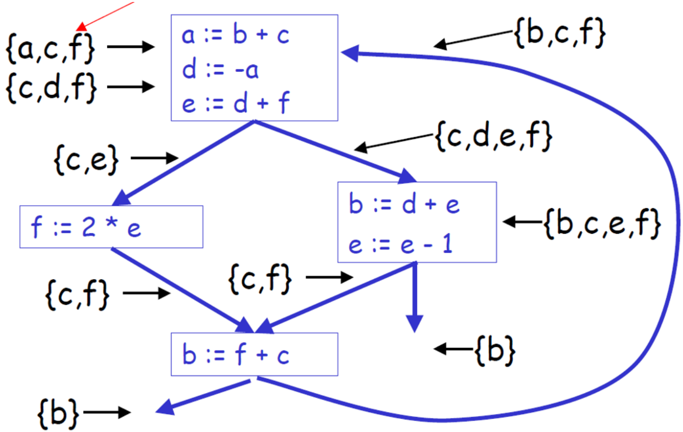
Final Code:

3 reg. are sufficient for no spills

r1 = 2
r2 = 4
r3 = r1 + r2
r3 = r1 + 1
r1 = r1 * r2
r2 = r3 * 2

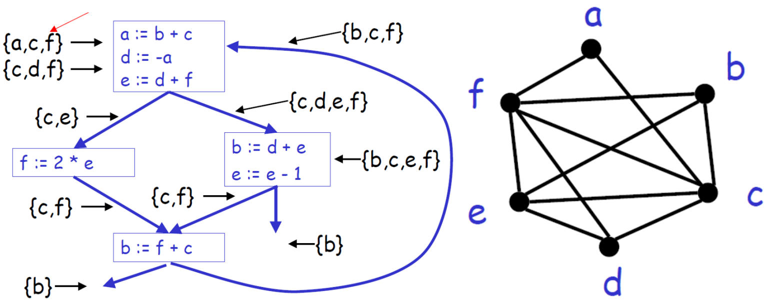
Another Example

Compute live variables at each point



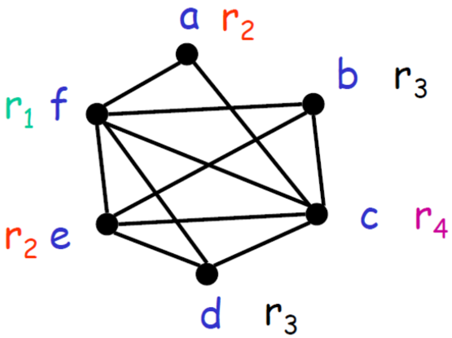
Register Interference Graph

- b and c cannot be in the same register
- b and d can be in the same register



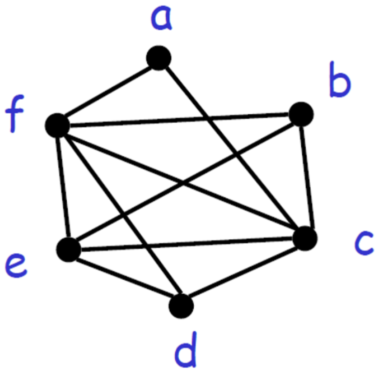
Graph Coloring: Example

- There is no coloring with less than 4 colors (has two 4-cliques)
- There are 4 colorings of the graph



Graph Coloring: Example

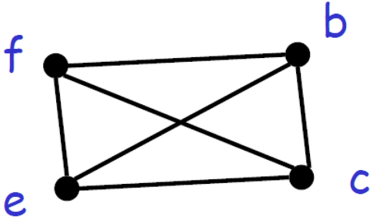
- Start with the RIG and with k = 4. Stack = {}



- Remove a and then d: Stack = {d, a}

Graph Coloring: Example

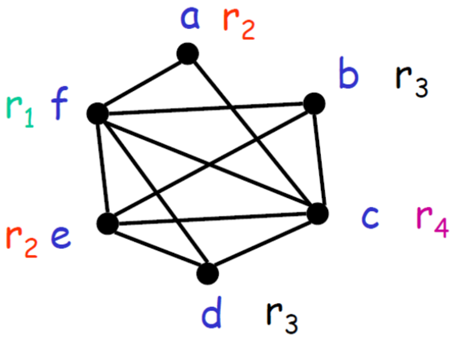
- Now all nodes have less than 4 neighbors and can be removed. Say, as: c, b, e, f



- Stack = {f, e, b, c, d, a}

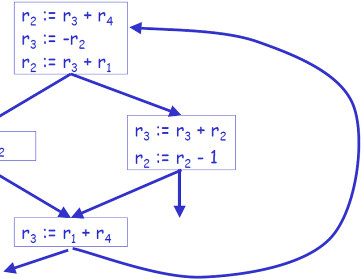
Graph Coloring: Example

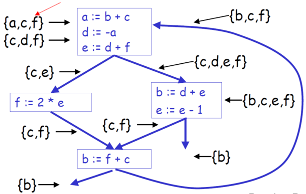
- Start assigning colors to: f, e, b, c, d, a

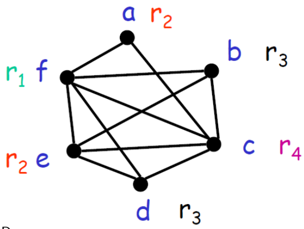


Code with Registers Allocated

- With the coloring the code becomes

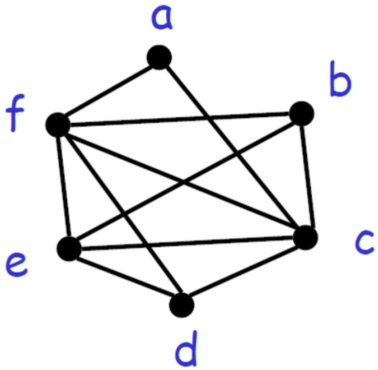






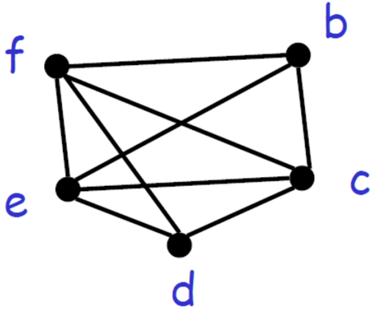
What if the Heuristic Fails?

- What if during simplification we get to a state where all nodes have k or more neighbors?
- Let us try a 3-coloring



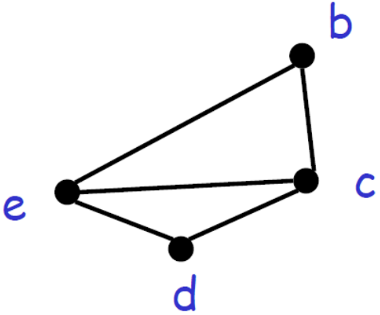
What if the Heuristic Fails?

- Remove a and get stuck
- Pick a node as a candidate for spilling
  - A spilled temporary "lives" in memory
- Assume that f is picked as a candidate



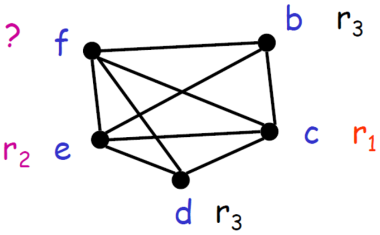
What if the Heuristic Fails?

- Remove f and continue the simplification
  - Simplification now succeeds: b, d, e, c



What if the Heuristic Fails?

- On the assignment phase we get to the point when we have to assign a color to f
- We hope that among the 4 neighbors of f we use less than 3 colors ⇒ **optimistic coloring**

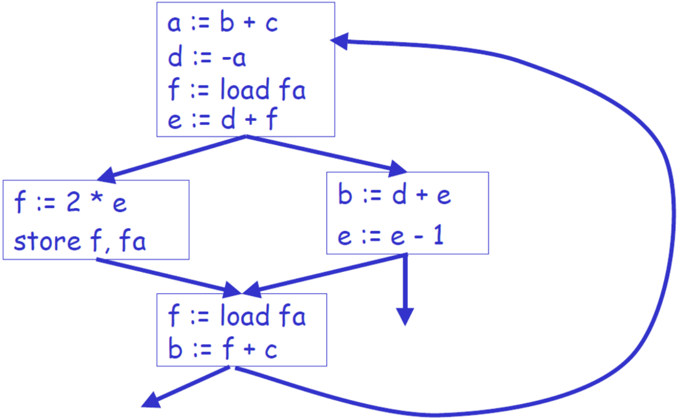


Spilling

- We fail and we must spill temporary f
- We must allocate a memory location as the home of f
  - Typically this is in the current stack frame
  - Call this address fa
- Before each operation that uses f, insert
  - f := load fa
- After each operation that defines f, insert
  - store f, fa

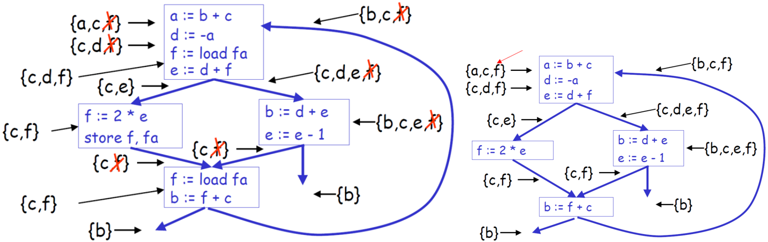
Code with Spilling

- The new code after spilling f



Recomputing Liveness Information

- The new liveness information after spilling

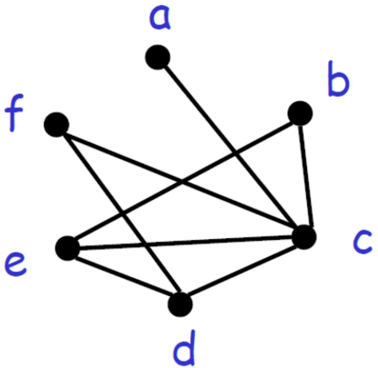


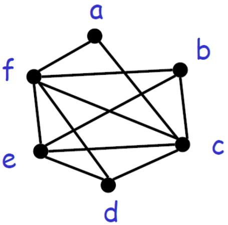
Recomputing Liveness Information

- The new liveness information is almost as before
- f is live only
  - Between a f := load fa and the next instruction
  - Between a store f, fa and the preceding instruction
- Spilling reduces the live range of f
- And thus reduces its interferences
- Which results in fewer neighbors in RIG for f

Recompute RIG after Spilling

- The only changes are in removing some of the edges of the spilled node
- In our case f still interferes only with c and d
- And the resulting RIG is 3-colorable





Spilling

- Additional spills might be required before a coloring is found
- The tricky part is deciding what to spill
- Possible heuristics:
  - Spill temporaries with most conflicts
  - Spill temporaries with few definitions and uses
  - Avoid spilling in inner loops
- Any heuristic is correct