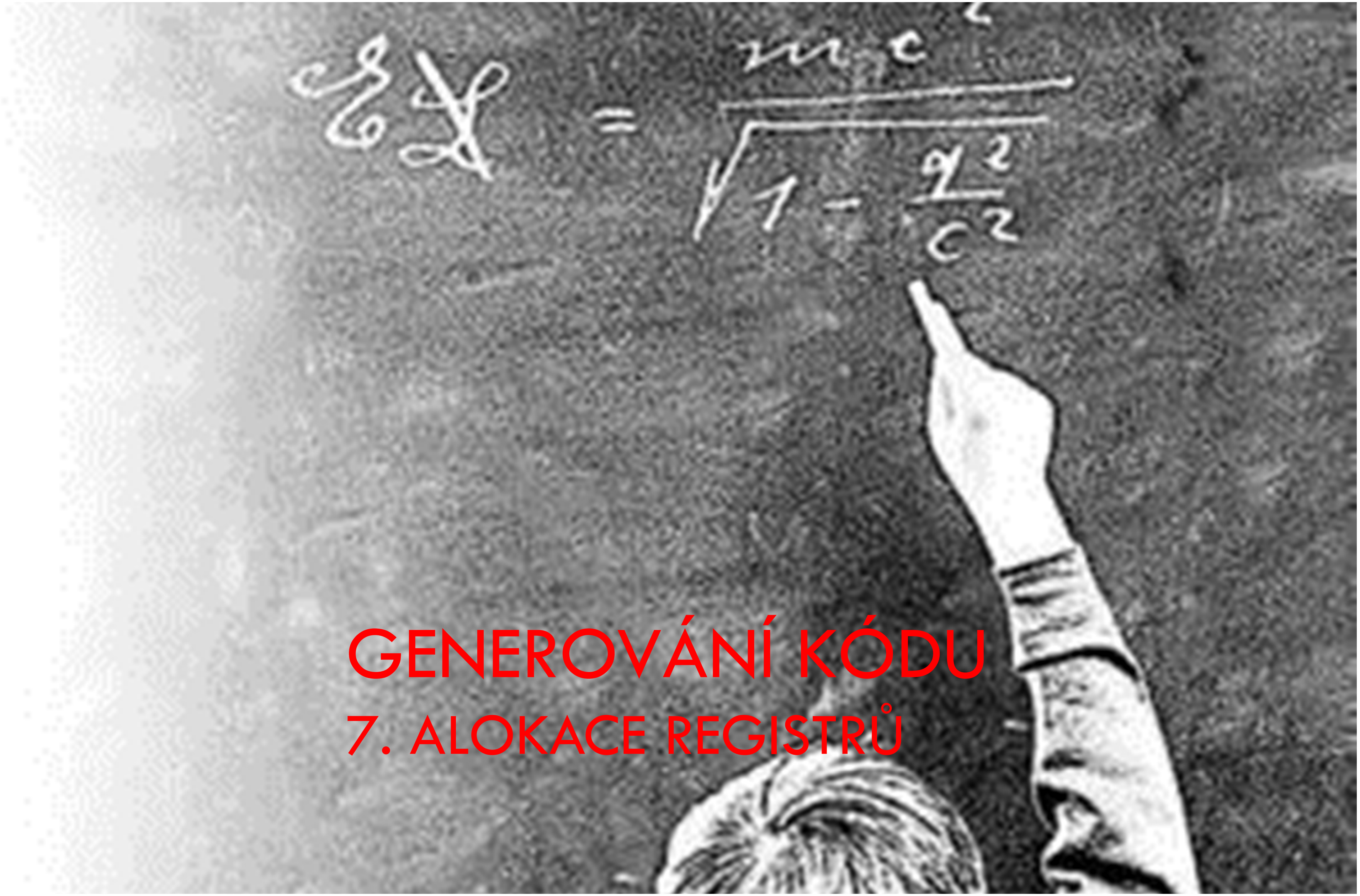


GENEROVÁNÍ KÓDU
7. ALOKACE REGISTRŮ

2011 Jan Janoušek
MI-GEN

OPP A
PRAHA PRAGUE PRAGA PRAG
EVROPSKÁ UNIE
Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

REGISTERS ALLOCATION

The problem

- Instructions operating with registers are significantly faster than those operating with memory

- IRs use as many temporaries as necessary
  - This complicates final translation to assembly
  - But simplifies code generation and optimization

- The allocation of registers
  - can be used in various back-end phases,
  - can be performed by various methods

- General rule: certain values, such as stack pointers, base registers,… are typically held in registers. Maximum of other values must be allocated to the remaining registers.

History of register allocation problem

- Register allocation is as old as intermediate code

- Register allocation was used in the original FORTRAN compiler in the '50s
  - Very crude algorithms

- A breakthrough was not achieved until 1980 when **Chaitin** invented a register allocation scheme based on graph coloring
  - Relatively simple, global and works well in practice

An Example of register allocation

- Consider the program

```
a := c + d
e := a + b
f := e - 1
```

  - with the assumption that `a` and `e` **die after use**

- Three states of temporaries:
  - **Unallocated –** temporary has not been assigned yet
  - **Live –** temporary allocated and will be used in future
  - **Dead –** temporary will not be used anymore

An Example of register allocation

- Consider the program

```
a := c + d
e := a + b
f := e - 1
```

  - with the assumption that `a` and `e` **die after use**

- Temporary `a` can be "reused" after `e := a + b`

- The same - Temporary `e` can be reuses after `f := e – 1`

- Can allocate `a`, `e`, and `f` all to one register (r_1):

$$r_1 := r_2 + r_3$$
$$r_1 := r_1 + r_4$$
$$r_1 := r_1 - 1$$

Basic Register Allocation Idea

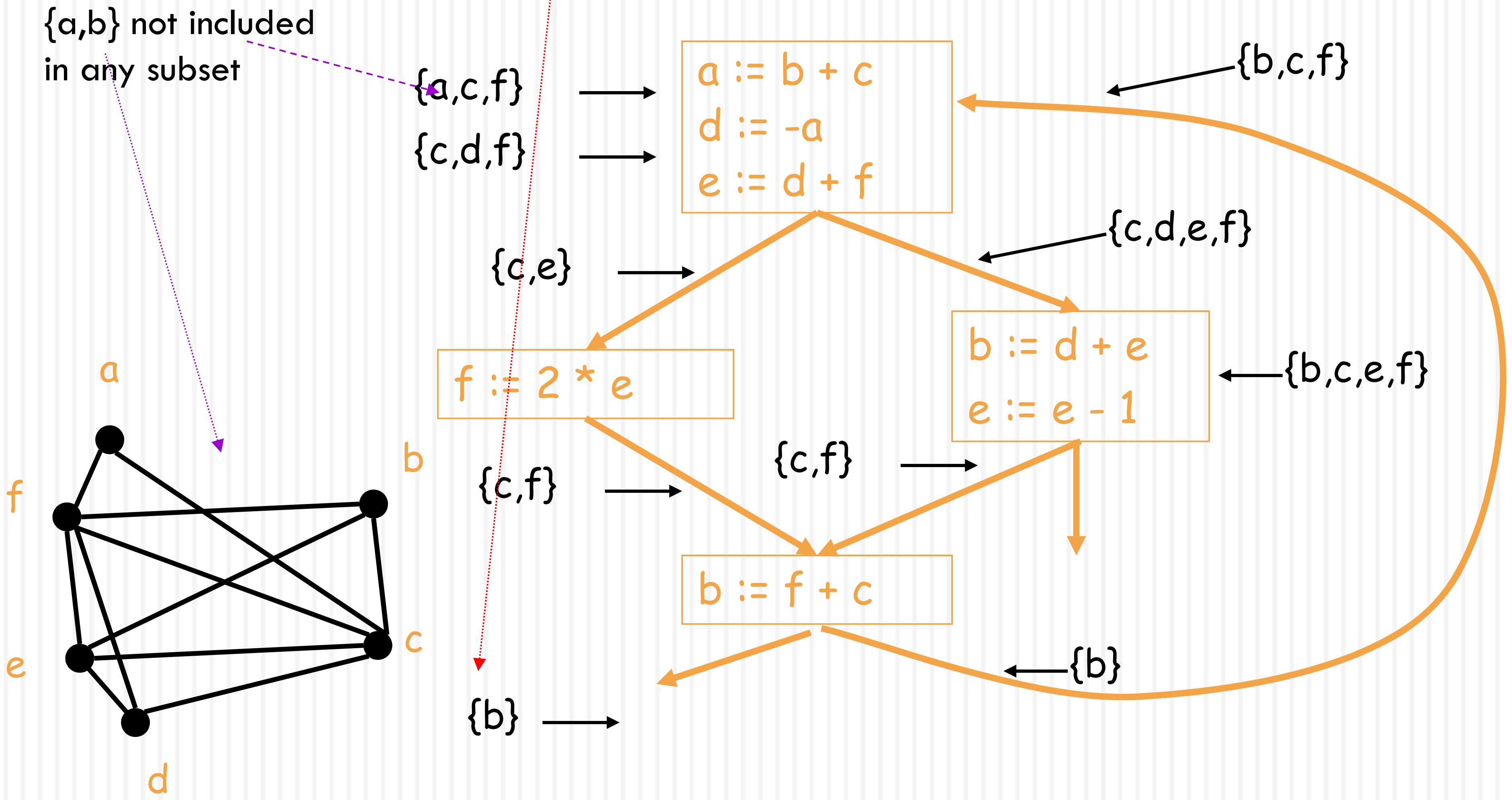
- The value in a dead temporary is not needed for the rest of the computation
  - A dead temporary can be reused

- **Basic rule:**
  - *Temporaries t_1 and t_2 can share the same register if* **at any point in the program at most one** *of t_1 or t_2 is live !*

Algorithm to minimize the number of registers: Part I

Example of basic blocks and flow graph

Compute live variables for each point:



The Register Interference Graph

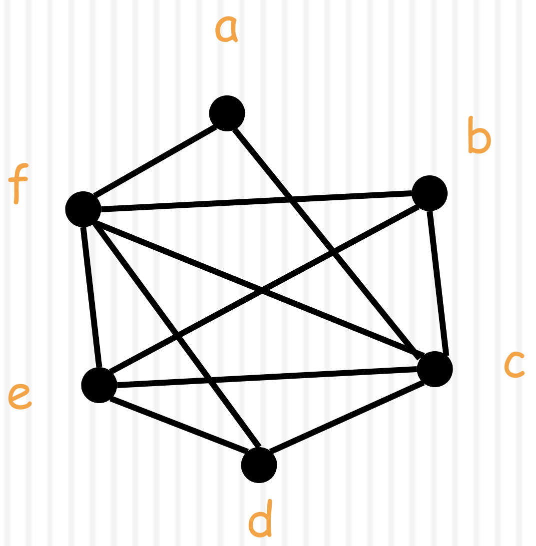
- Two temporaries that are live simultaneously cannot be allocated in the **same register**

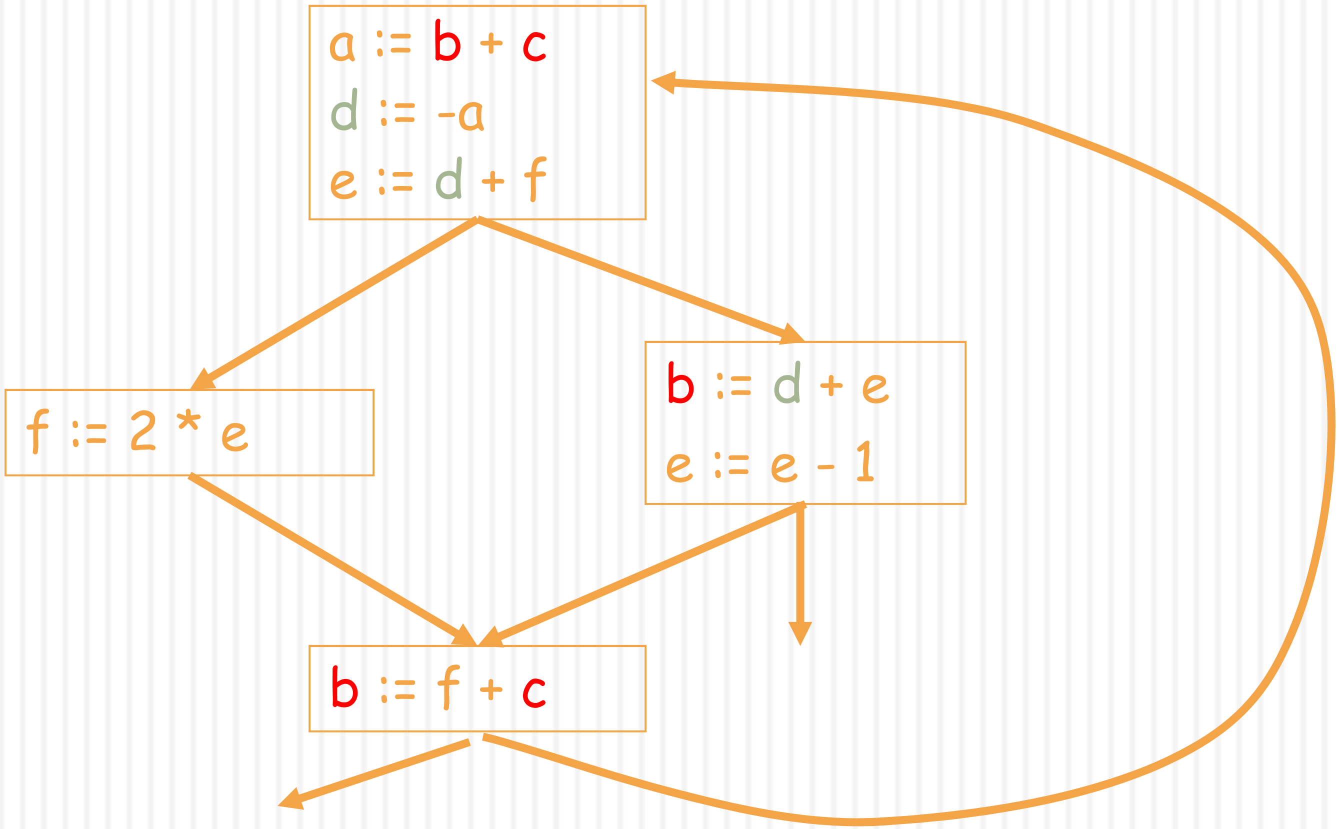
- We construct an undirected graph
  - A node for each temporary
  - An edge between t_1 and t_2 if they are live simultaneously at some point in the program

- This is the register interference graph (RIG)
  - Two temporaries can be allocated to the same register **if there is no edge connecting them**

Register Interference Graph. Example.

For our example:





- E.g., b and c cannot be in the same register
- E.g., b and d can be in the same register

Properties of Register Interference Graph.

- It extracts exactly the information needed to **characterize legal register assignments**

- It gives a **global** (i.e., over the entire flow graph) picture of the **register requirements**

- After RIG construction the register allocation algorithm is **architecture independent**

Graph Coloring. Definitions.

- A **coloring of a graph** is an assignment of colors to nodes, such that nodes connected by an edge have different colors
- A graph is **k-colorable** if it has a coloring with k colors

Register Allocation Through Graph Coloring

- Rewrite the code (generated from IR for example) which uses unrestricted number of registers so that it would use only real machine registers.

Register Allocation Through Graph Coloring

- In our problem, **colors = registers**
  - We need to assign colors (registers) to graph nodes (temporaries)

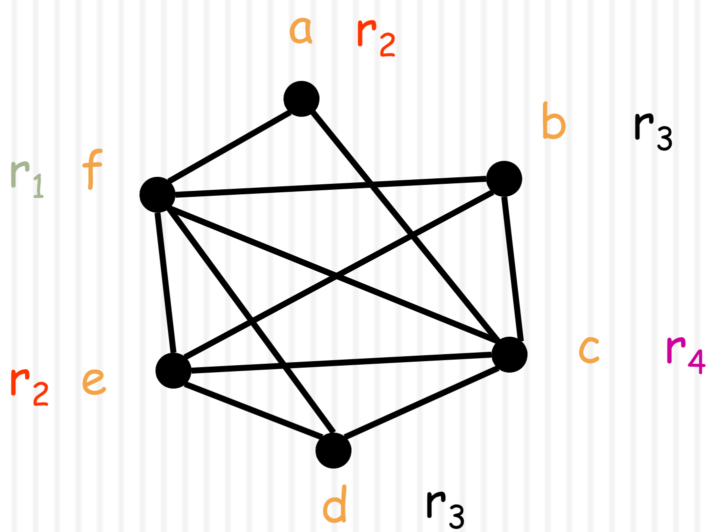
- Let k = number of machine registers

- If the RIG is **k-colorable** then there is a register assignment that uses **no more than k registers**

Register Interference Graph

Graph Coloring. Example.

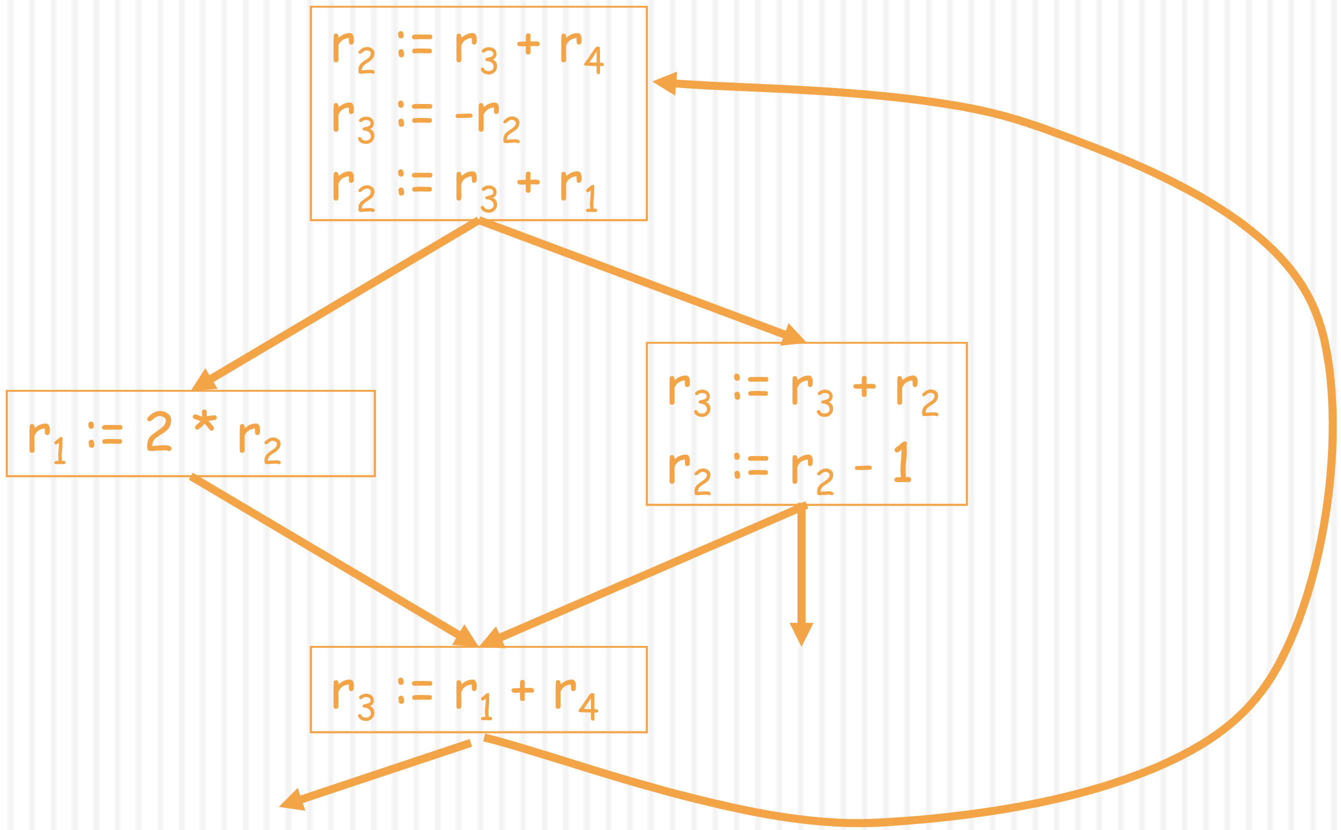
Consider the example RIG



- There is no coloring with less than 4 colors
- There are 4-colorings of this graph

Graph Coloring. Example.

Under this coloring the code becomes:



Computing Graph Colorings

- The remaining problem is to compute a coloring for the interference graph
- But:
  - This problem is very hard (NP-hard). No efficient algorithms are known.
  - A coloring might not exist for a given number or registers
- The solution to is to use heuristics

Graph Coloring Heuristic

- Observation:
  - Pick a node t with fewer than k neighbors in RIG
  - Eliminate t and its edges from RIG
  - If the resulting graph has a k-coloring then so does the original graph

- Why:
  - Let $c_1,\ldots,c_n$ be the colors assigned to the neighbors of t in the reduced graph
  - Since $n < k$ we can pick some color for t that is different from those of its neighbors

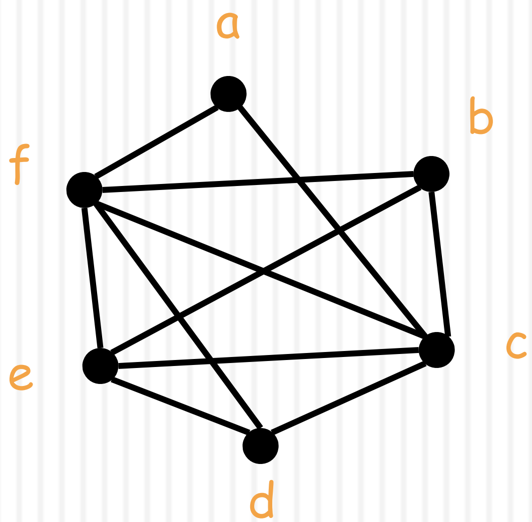
Graph Coloring Heuristic

- The following works well in practice:
  - Pick a node t with fewer than k neighbors
  - Put t on a stack and remove it from the RIG
  - Repeat until the graph has one node

- Then start assigning colors to nodes on the stack (starting with the last node added)
  - At each step pick a color different from those assigned to already colored neighbors

Graph Coloring Example (1)

Start with the RIG and with k = 4:

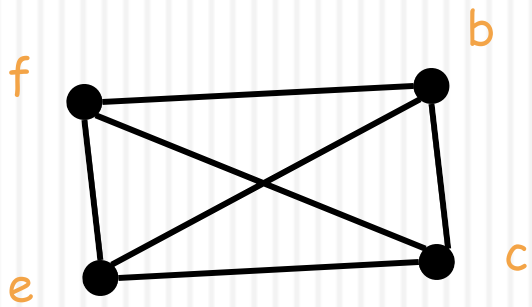


Stack: {}

- Remove a and then d

Graph Coloring Example (2)

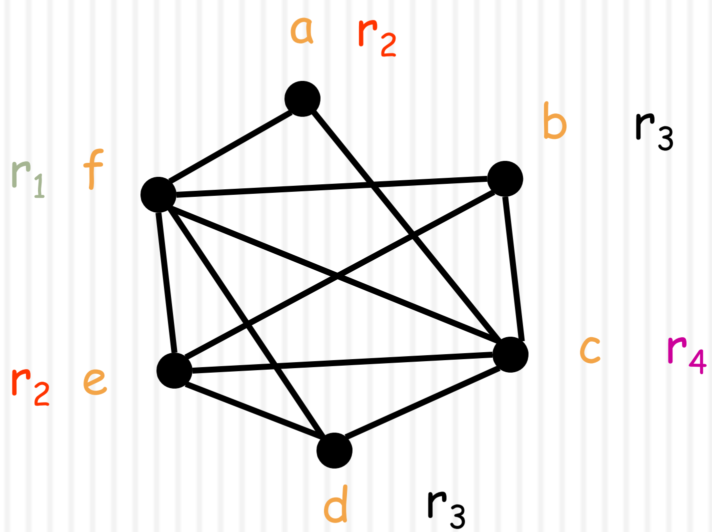
Now all nodes have fewer than 4 neighbors and can be removed: c, b, e, f



Stack: {d, a}

Graph Coloring Example (2)

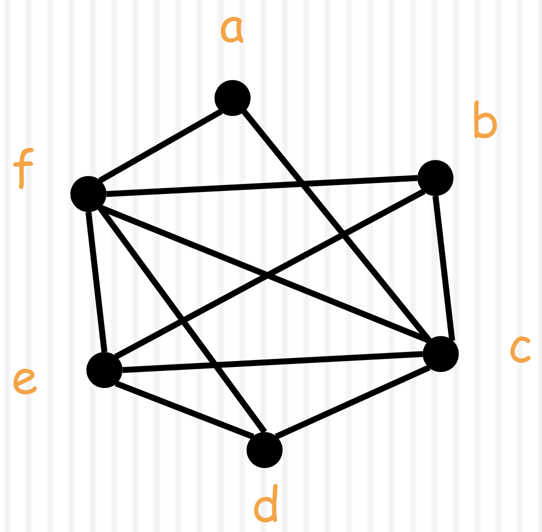
Start assigning colors to: f, e, b, c, d, a



What if the Heuristic Fails?

- What if during simplification we get to a state where all nodes have k or more neighbors ?

- **Example:** try to find a 3-coloring of the RIG:

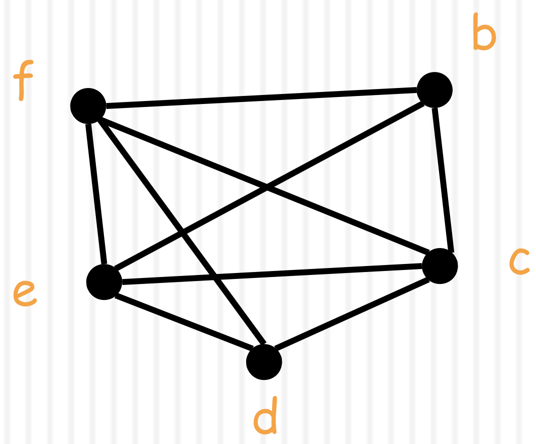


What if the Heuristic Fails?

Remove a and get stuck (as shown below)

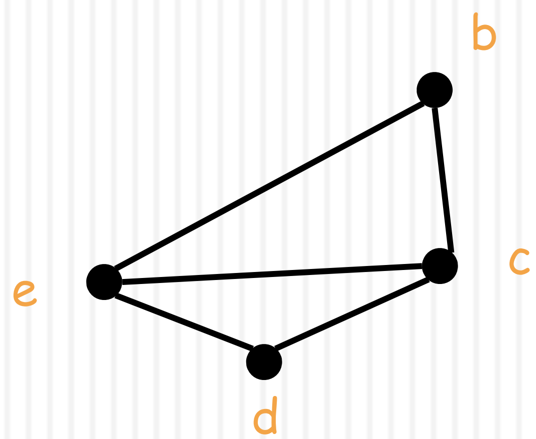
- Pick a node as a candidate for **spilling**
  - **A spilled temporary "lives" in memory**

- Assume that f is picked as a candidate



What if the Heuristic Fails?

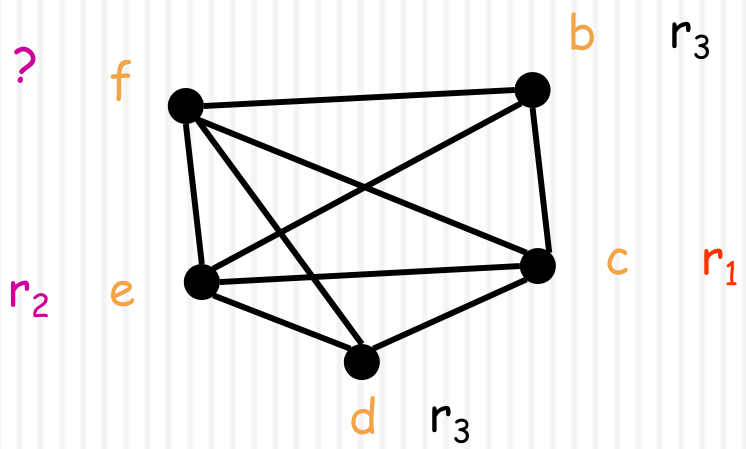
- Remove f and continue the simplification
  - Simplification now succeeds: b, d, e, c



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 $\Rightarrow$ **optimistic coloring**

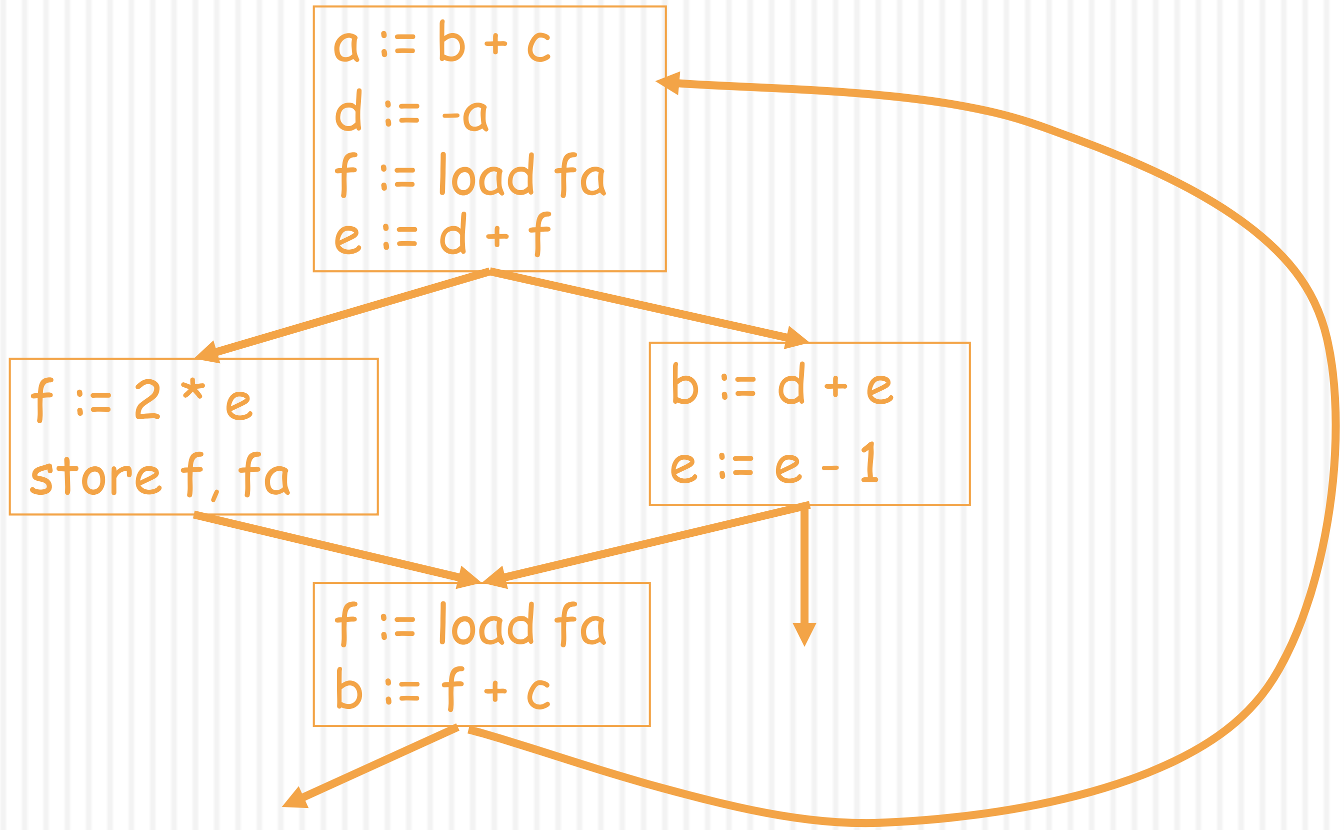


Spilling

- Since optimistic coloring failed 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`

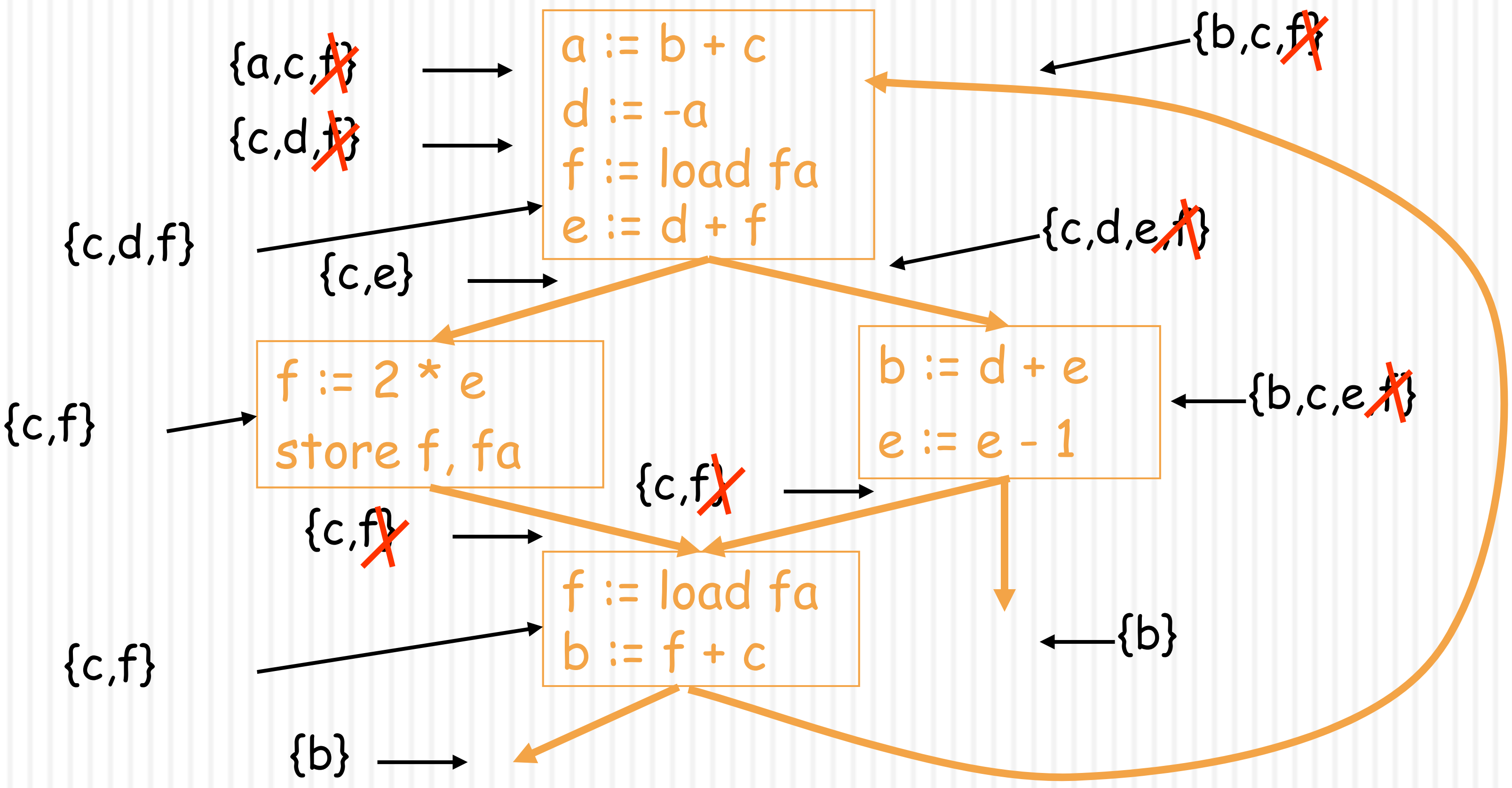
Spilling. Example.

This is the new code after spilling f



Recomputing Liveness Information

The new liveness information after spilling:

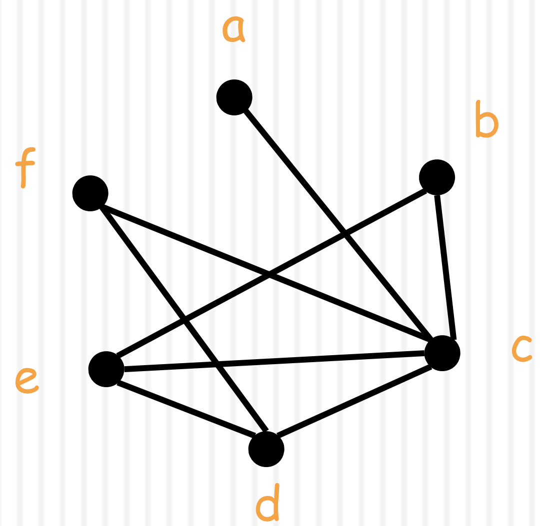


Recomputing Liveness Information

- 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 result 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 (Cont.)

- 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
  - **General rule: try to avoid spilling in inner loops**

- Any heuristic is correct

Linear scan register allocation

- simpler but faster

- widely used in JIT compilers

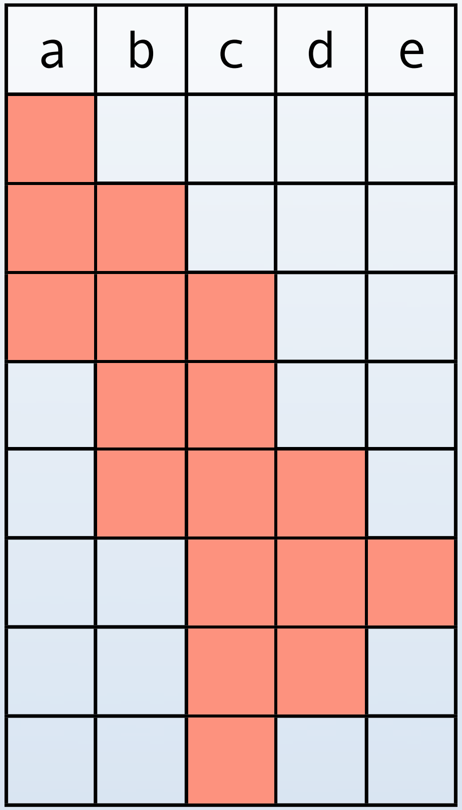
Linear scan algorithm

Linear scan works on a linear representation of the program. Live ranges must be known for all values.

The algorithm scans live ranges from first to last. Whenever there are less than K values live at the same time, they are all put in registers. When all registers are allocated and a new value becomes live, one of them must be spilled. The one whose live range ends last is systematically chosen.

Linear scan example

Live ranges



Allocation

R1	R2
a	
a	b
a	b
	b
d	b
d	e
d	

c is spilled

Linear scan and spilling

When values are spilled to memory, some registers must be available to operate on them – at least on modern processors that cannot operate on values stored in memory.

There are two ways to make sure that these registers are available:

1. reserve them in advance, which can be sub-optimal if no values are spilled,
2. perform the allocation without reserving them; if spilling turns out to be required, reserve spilling registers and redo the allocation.

Notes as conclusion

- Register allocation is a “must have” optimization in most compilers:
  - Because intermediate code uses too many temporaries
  - Because it makes a big difference in performance
- Graph coloring is a powerful register allocation schemes
- Register allocation is more complicated for CISC machines
- Graph coloring is useful in standard compilers, sometimes it is too slow. Faster methods, such as a linear register scan, are used for example in Just-In-Time compilers (Java).