CS536

Register Allocation

A Sahu

CSE, IIT Guwahati

Outline

- Register Allocation
- Spilling
- Spilling Example with RA

Register Allocation

How to best use the bounded number of registers.

- Reducing load/store operations
- What are best values to keep in registers?
- When can we ‘free’ registers?

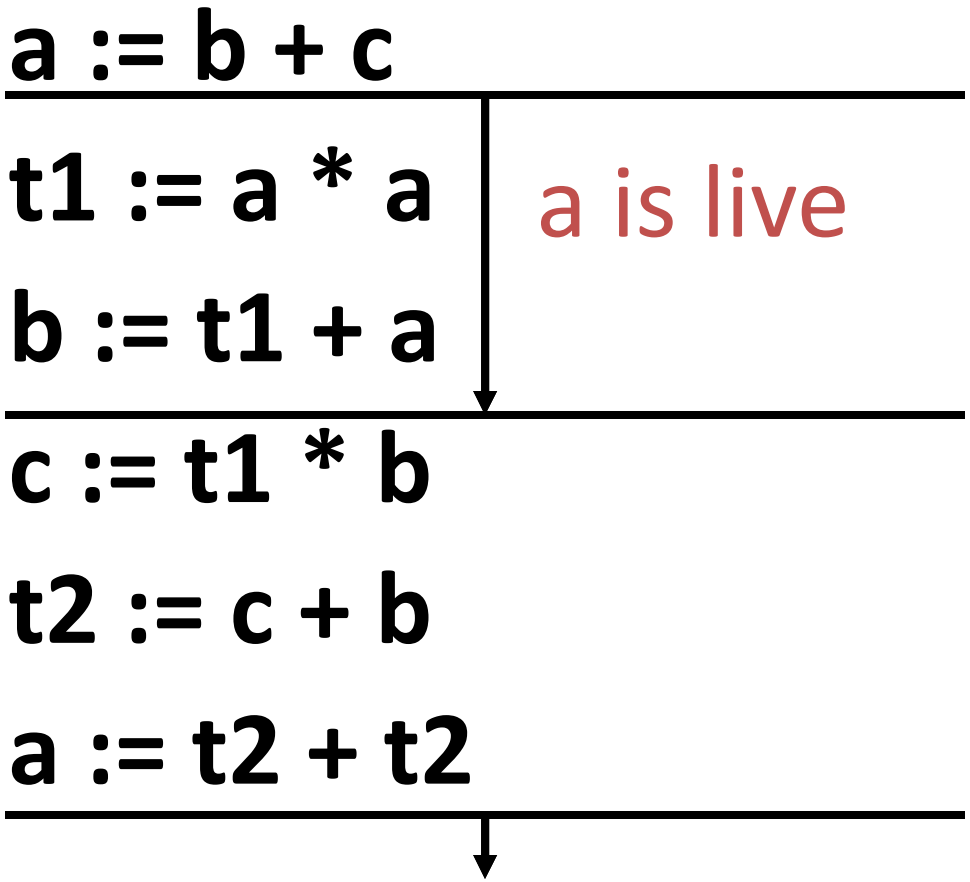
Complications:

- special purpose registers
- operators requiring multiple registers.

Register Allocation Algorithms

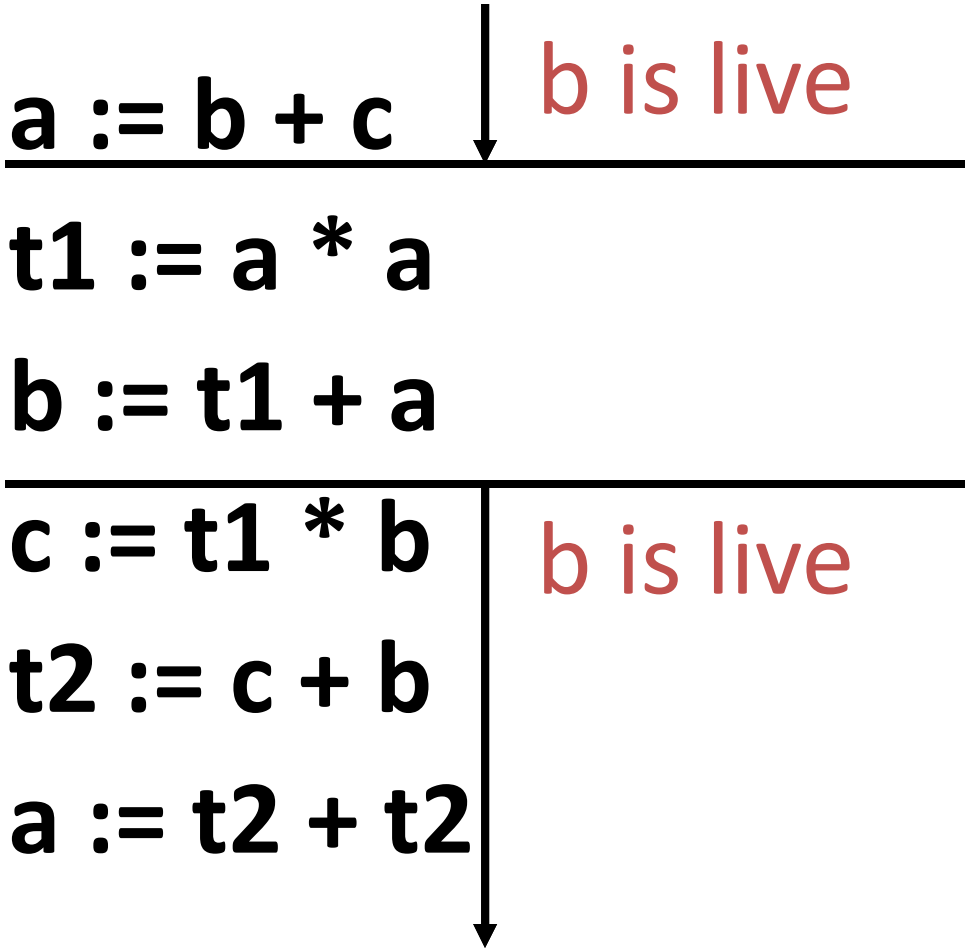
- Local (basic block level):
  - **Basic - using liveness information**
  - **Register Allocation using graph coloring**
- Global (CFG)
  - Need to use global liveness information

Example: When is a live?



Assume a,b and c are used after this basic block

Example: When is b live?



Assume a,b and c are used after this basic block

Computing live status in basic blocks

Input: A basic block.

Output: For each statement, set of live variables

1. Initially all non-temporary variables go into live set (L).
2. for i = *last* statement to *first* statement:

   for statement i: x := y op z

   1. Attach L to statement i.
   2. Remove x from set L.
   3. Add y and z to set L.

Example Answers

live = {b,c} ← what does this mean???

a := b + c

live = {a}

t1 := a * a

live = {a,t1}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

A Code Generator

- Generates target code for a sequence of three-address statements using next-use information
- Uses new function ***getreg*** to assign registers to variables
- Computed results are kept in registers as long as possible, which means:
  - Result is needed in another computation
  - Register is kept up to a procedure call or end of block
- Checks if operands to three-address code are available in registers

The Code Generation Algorithm

- For each statement *x* := *y* op *z*
  1. Set location *L* = *getreg*(*y*, *z*)
  2. If *y* ∉ *L* then generate
     MOV *y'*,*L*
     where *y'* denotes one of the locations where the value of *y* is available (choose register if possible)
  3. Generate
     OP *z'*,*L*
     where *z'* is one of the locations of *z*;
     Update register/address descriptor of *x* to include *L*
  4. If *y* and/or *z* has no next use and is stored in register, update register descriptors to remove *y* and/or *z*

Register and Address Descriptors

- A ***register descriptor*** keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

  `MOV a,R0` "**`R0`** contains **`a`**"

- An ***address descriptor*** keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

  `MOV a,R0`
  `MOV R0,R1` "**`a`** in **`R0`** and **`R1`**"

The *getreg* Algorithm

- To compute *getreg*(*y*,*z*)
  1. If *y* is stored in a register *R* and *R* only holds the value *y*, and *y* has no next use, then return *R*;
     Update address desc.: value *y* no longer in *R*
  2. Else, return a new empty register if available
  3. Else, find an occupied register *R*;
     Store contents (register spill) by generating
     `MOV` *R*, *M*
     for every *M* in address descriptor of *y*;
     Return register *R*
  4. Return a memory location

Code Generation Example

Statements	Code Generated	Register Descriptor	Address Descriptor
		Registers empty	
t:= a - b	MOV a,R0 SUB b,R0	R0 contains t	t in R0
u:= a - c	MOV a,R1 SUB c,R1	R0 contains t R1 contains u	t in R0 u in R1
v:= t + u	ADD R1,R0	R0 contains v R1 contains u	u in R1 v in R0
d:= v + u	ADD R1,R0 MOV R0,d	R0 contains d	d in R0 d in R0 and memory

Register Allocation and Assignment

- The *getreg* algorithm is simple but sub-optimal
  - All live variables in registers are stored (flushed) at the end of a block
- *Global register allocation* assigns variables to limited number of available registers and attempts to keep these registers consistent across basic block boundaries
  - Keeping variables in registers in looping code can result in big savings

Allocating Registers in Loops

- Suppose loading a variable *x* has a cost of 2
- Suppose storing a variable *x* has a cost of 2

- Benefit of allocating a register to a variable *x* within a loop *L* is

  $$\sum_{B \in L} (use(x, B) + 2\ live(x, B))$$

  where *use*(*x*, *B*) is the number of times *x* is used in *B* and *live*(*x*, *B*) = true if *x* is live on exit from *B*

The Register Allocation Problem

- Motivation: we want to hold all the temporary values in registers
- Recall that intermediate code uses as many variables as necessary
  - This complicates final translation to assembly
  - But simplifies code generation and optimization
  - Typical intermediate code uses too many variables
- The register allocation problem:
  - Rewrite the intermediate code to use fewer variables than there are machine registers
  - Method: assign more variables to the same register
    - But without changing the program behavior

An Example

- Consider the program
  - with the assumption that a and e die after use

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

- Variable a can be "reused" after "a + b"
- Same with variable e after "e - 1"
- So, we can allocate a, e, and f all to one register (r_1):

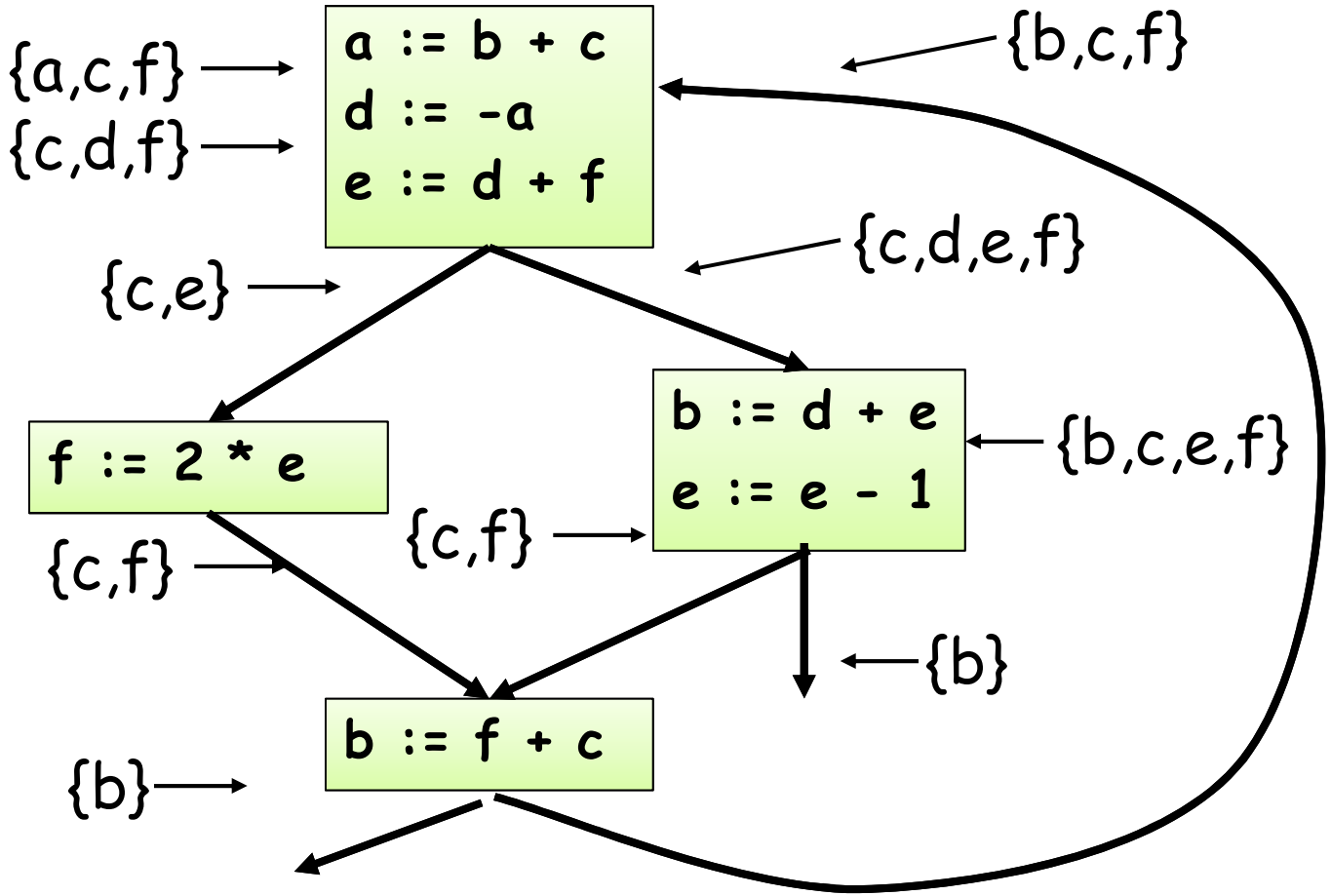
```
r1 = c + d
r1 = r1 + b
r1 = r1 - 1
```

Basic Register Allocation Idea

- The value in a dead variable is not needed for the rest of the computation
  - A dead temporary can be reused
- Basic rule:
  - Variables 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 alive !

Algorithm: Part I

- Compute live variables for each point:

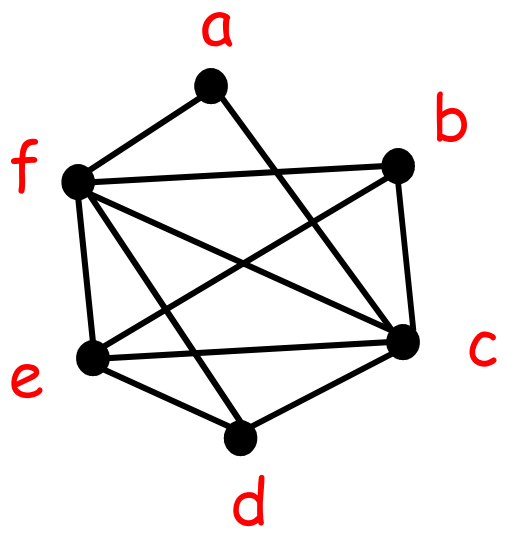


The Register Interference Graph

- Two variables that are alive 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 are not connected since there is no set of live variables that contains both b and d. Thus, they can be assigned to the same register

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

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

Problems with Graph coloring approach

- This is a well known NP-complete problem
  - We need to use heuristics
- What's happen if we cannot color the graph with K colors (assuming K is the number of the general purpose architectural registers
  - We need to use "Register Spilling"

Spilling

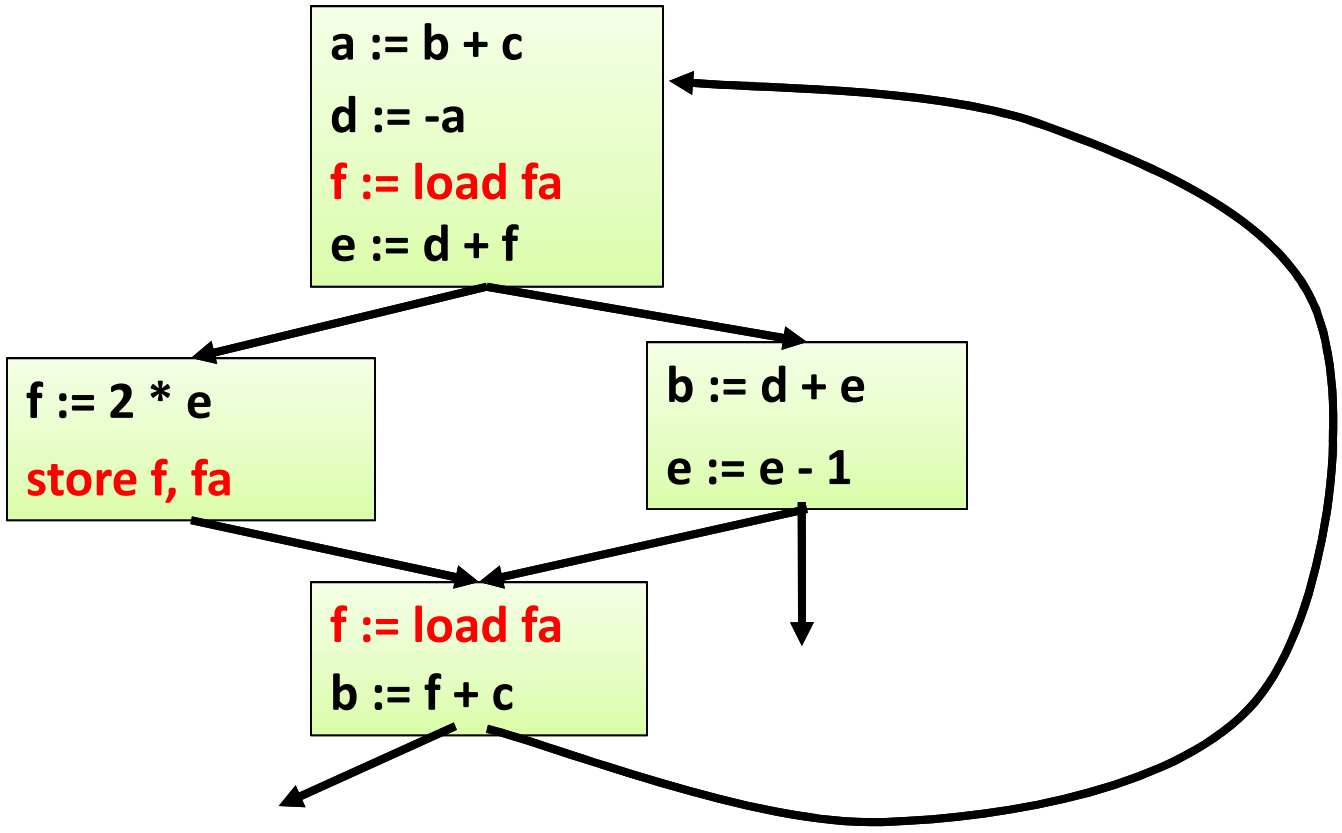
- Since optimistic coloring failed we must spill variable f
- We must allocate a memory location as the home of f
  - Typically this is in the current stack frame
  - Call this address **fa**
- Before each operation that uses **f**, insert

  f := load fa
- After each operation that defines **f**, insert

  store f, fa

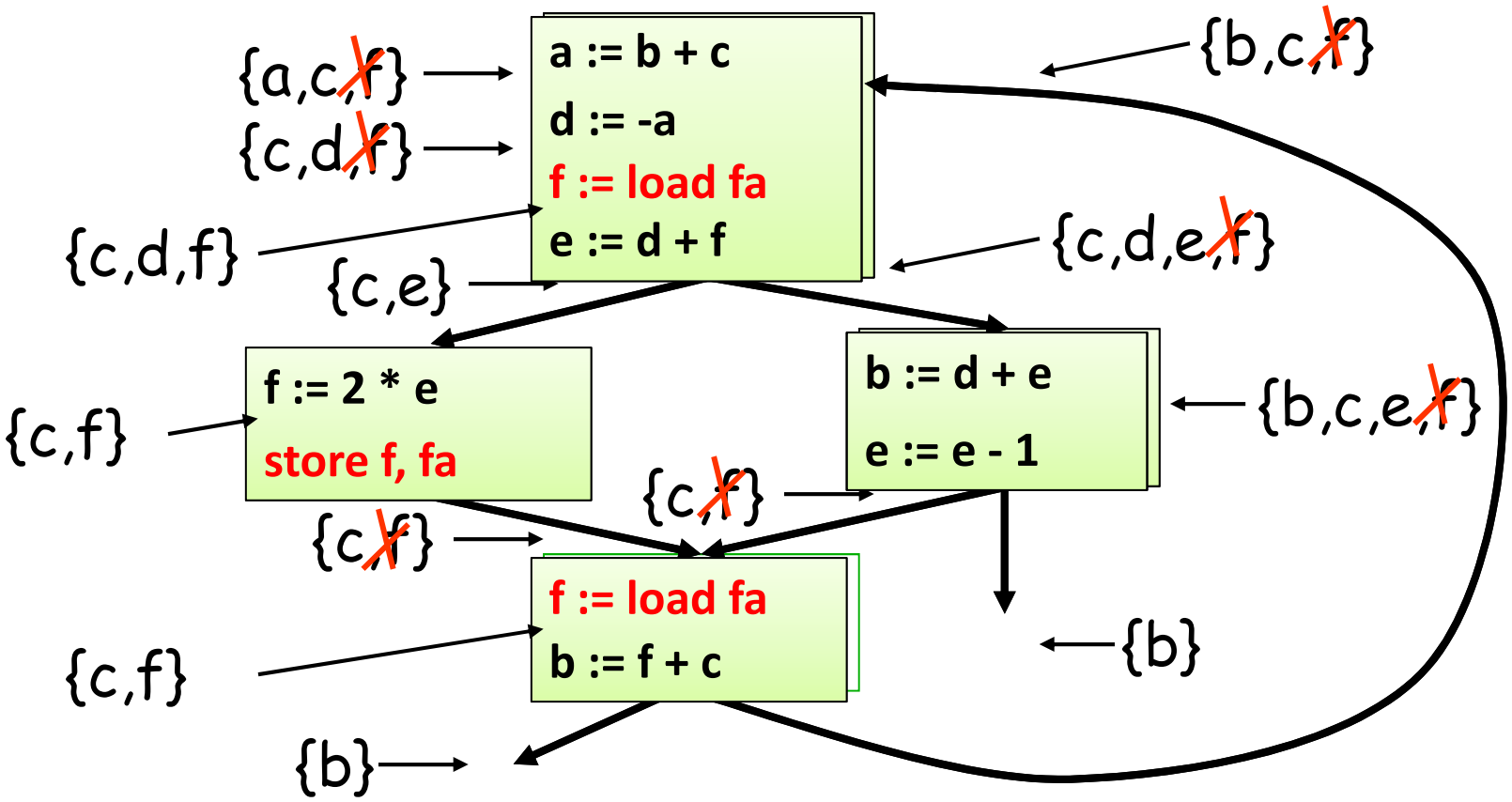
Spilling. Example.

- 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 instr.
- Spilling reduces the live range of **f**
- This allow to use the same architectural register for both f and other variable(s)
- If we can allocate all the variable to existing register, the procedure is done. Otherwise we need to spill another variable(s).