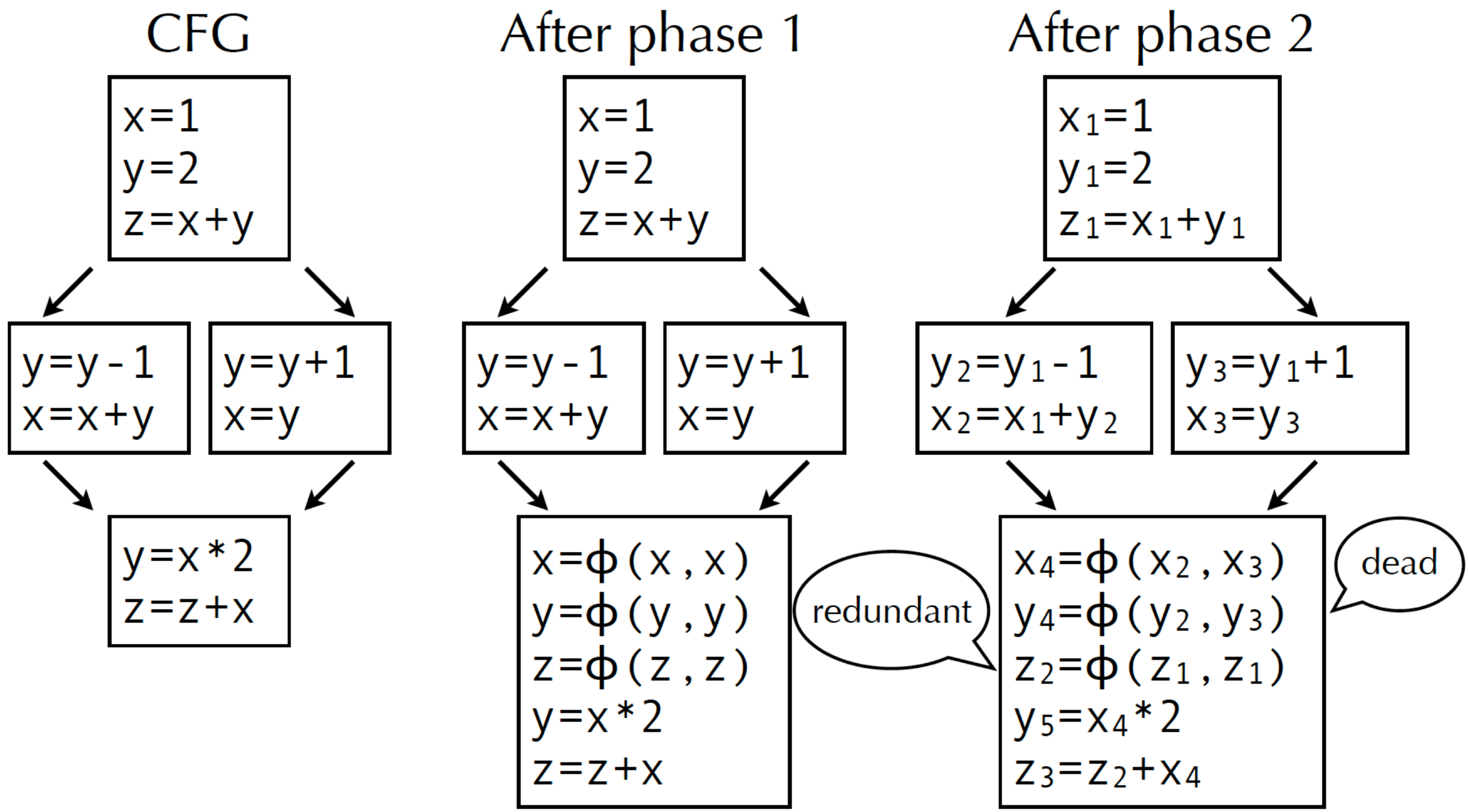
CS293S
SSA & Dead Code Elimination

Yufei Ding

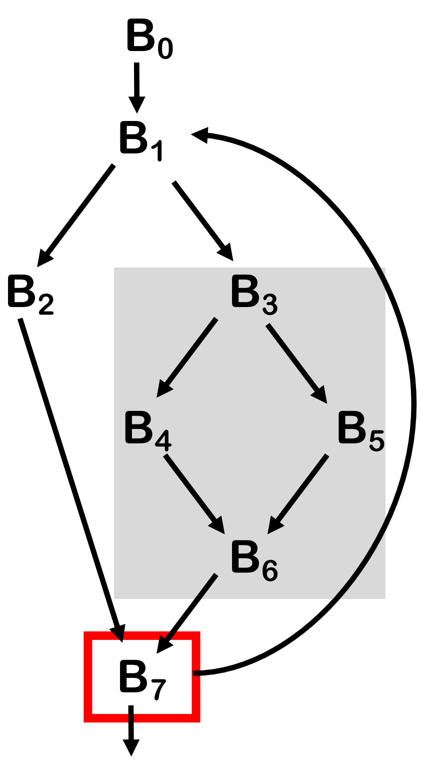
Review of Last Class

- Two other flow analysis (DFA)
  - Constant Propagation
  - Reaching definitions (def-use chain)
- Static Single Assignment(SSA)
  - Maximal SSA (all variables in every joint block)
  - Minimal SSA (a def in block n results in an insertion in each of its DF(n))
  - Semi-pruned SSA (similar as minimal SSA, but only focus on global variable definitions)

Maximal SSA


CFG
x=1
y=2
z=x+y
y=y-1
x=x+y
y=y+1
x=y
y=x*2
z=z+x
After phase 1
x=1
y=2
z=x+y
y=y-1
x=x+y
y=y+1
x=y
x=ϕ(x,x)
y=ϕ(y,y)
z=ϕ(z,z)
y=x*2
z=z+x
redundant
After phase 2
x1=1
y1=2
z1=x1+y1
y2=y1-1
x2=x1+y2
y3=y1+1
x3=y3
x4=ϕ(x2,x3)
y4=ϕ(y2,y3)
z2=ϕ(z1,z1)
y5=x4*2
z3=z2+x4
dead

Dominance Frontiers



Dominance frontier of 3={7}

Dominance Frontiers

- **DF(n) is fringe just beyond the region n dominates**
- **m ∈DF(n) : iff n ∉(Dom(m) - {m}) but n ∈ DOM(p) for some p ∈ preds(m).**

i.e., n doesn't strictly dominate m

i.e., n dominates m's some parent

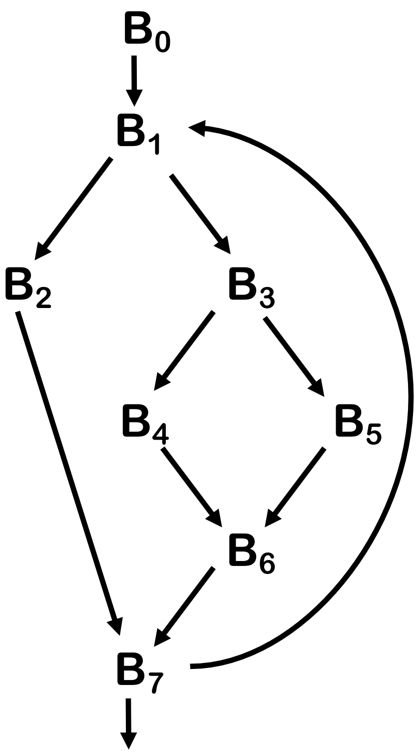
	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	–	1	7	7	6	6	7	1

A node can be the dominance frontier of itself.

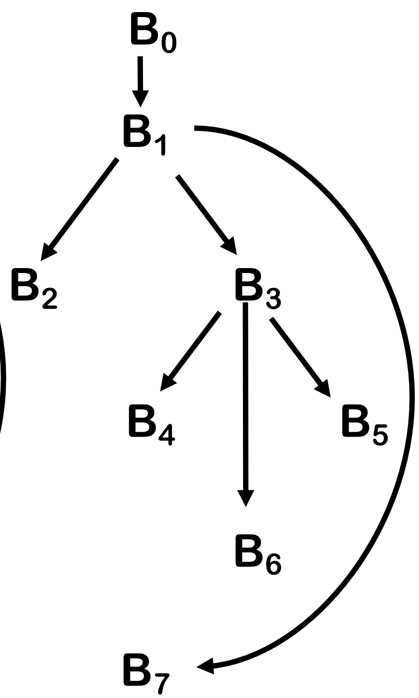
*** e.g., for node n = 1. It dominates its own parent, node 7, but does not directly dominate itself.**

*** it often indicates that there is a back edge.**

Computing Dominance Frontiers



Control flow graph (CFG)



Dominance Tree

- Only join points are in DF(n) for some n
- Leads to a simple, intuitive algorithm for computing dominance frontiers

  For each join point x

  For each CFG predecessor of x (from the CFG)

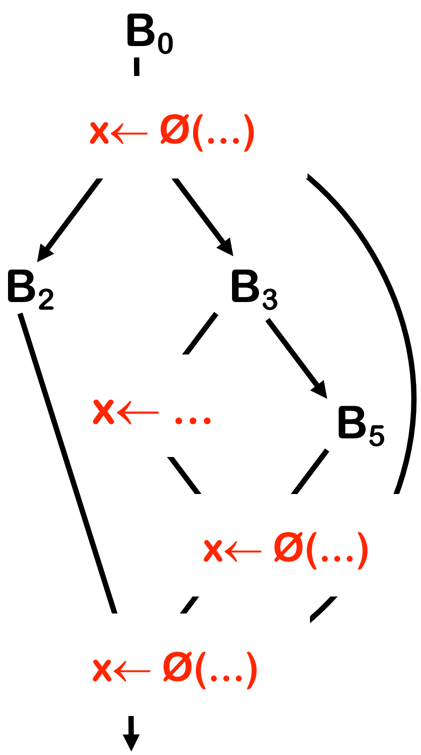
  Walk up to IDOM(x) in the dominator tree, adding x to DF(n) for each n in the walk except IDOM(x).

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	–	1	7	7	6	6	7	1

- What if there is another back edge from B_7 to B_0?

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	–	1	7	7	6	6	7	1

Minimal SSA: Ø-functions insertion with DF




```
for each variable x in the CFG
    work list = get all nodes (basic blocks) in which x is defined
    for each node n in work list
        for each node m in DF(n)
            if (there is no ϕ-function for x in m)
                insert a ϕ-function for x in m
                work list = work list ∪ { m }
```

	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	–	1	7	7	6	6	7	1

- **DF(4) is {6}, so ← in 4 forces Ø-function in 6**
- **← in 6 forces Ø-function in DF(6) = {7}**
- **← in 7 forces Ø-function in DF(7) = {1}**
- **← in 1 forces Ø-function in DF(1) = Ø (halt)**

Focus of This Class

- Static Single Assignment(SSA)
  - Semi-pruned SSA
    - (similar as minimal SSA, but on only global variable)
  - Pruned SSA
    - (similar as semi-pruned SSA, but dead are removed)

- Techniques for Removing φ-functions

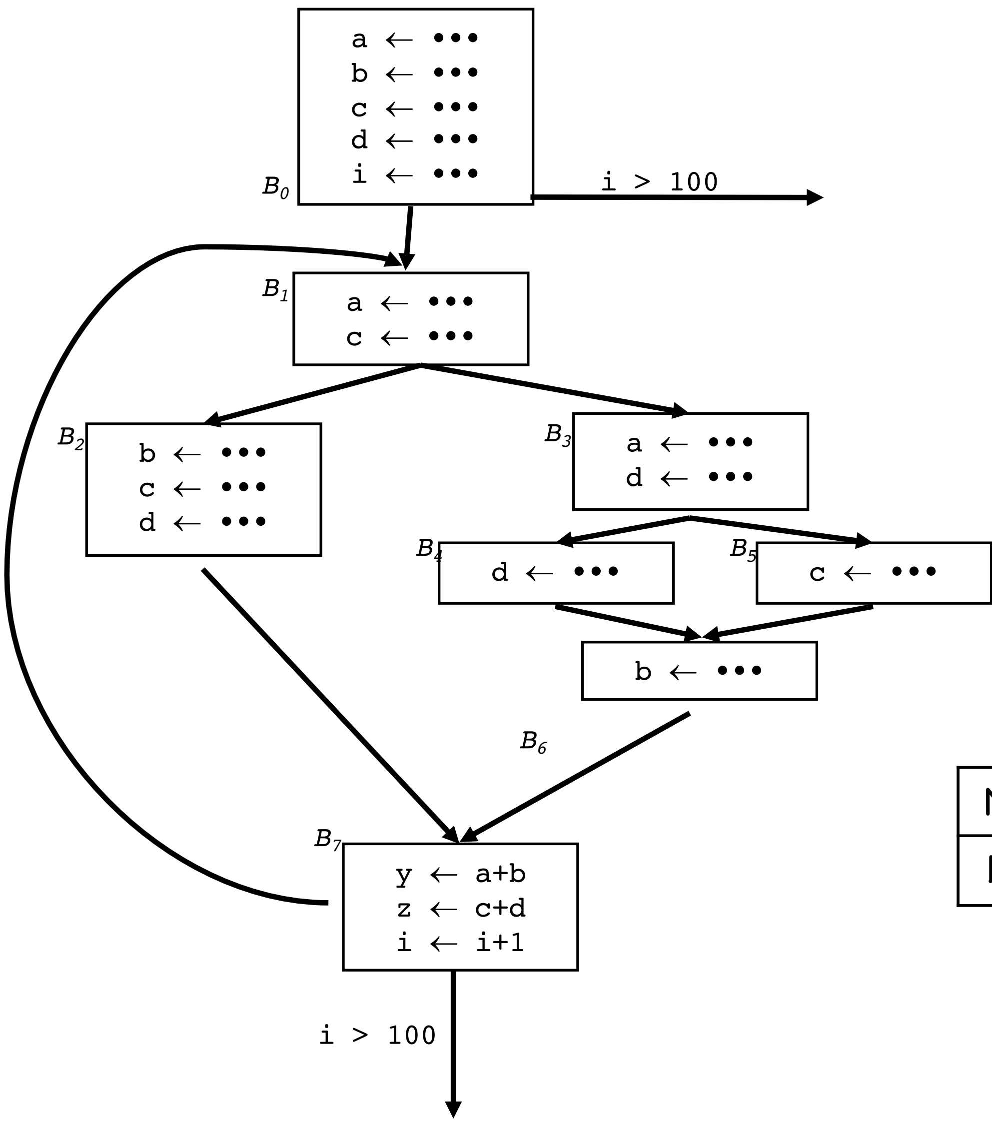
- Dead code elimiation

Semi-pruned SSA

- Observation: a variable that is only live in a single node can never have a live Ø-function.

- Therefore, the minimal technique can be further refined by first computing the set of global names – defined as the names that are live across more than one node – and producing Ø-functions for these names only.

Step 1: Global Variables



- Globals = $\cup_{all\ n\ in\ CFG}$ UEVar(n)

UEVar(n): Upper exposed variables in block n, i.e., variables used before it is redefined in block n. (We have learned this in liveness analysis.)

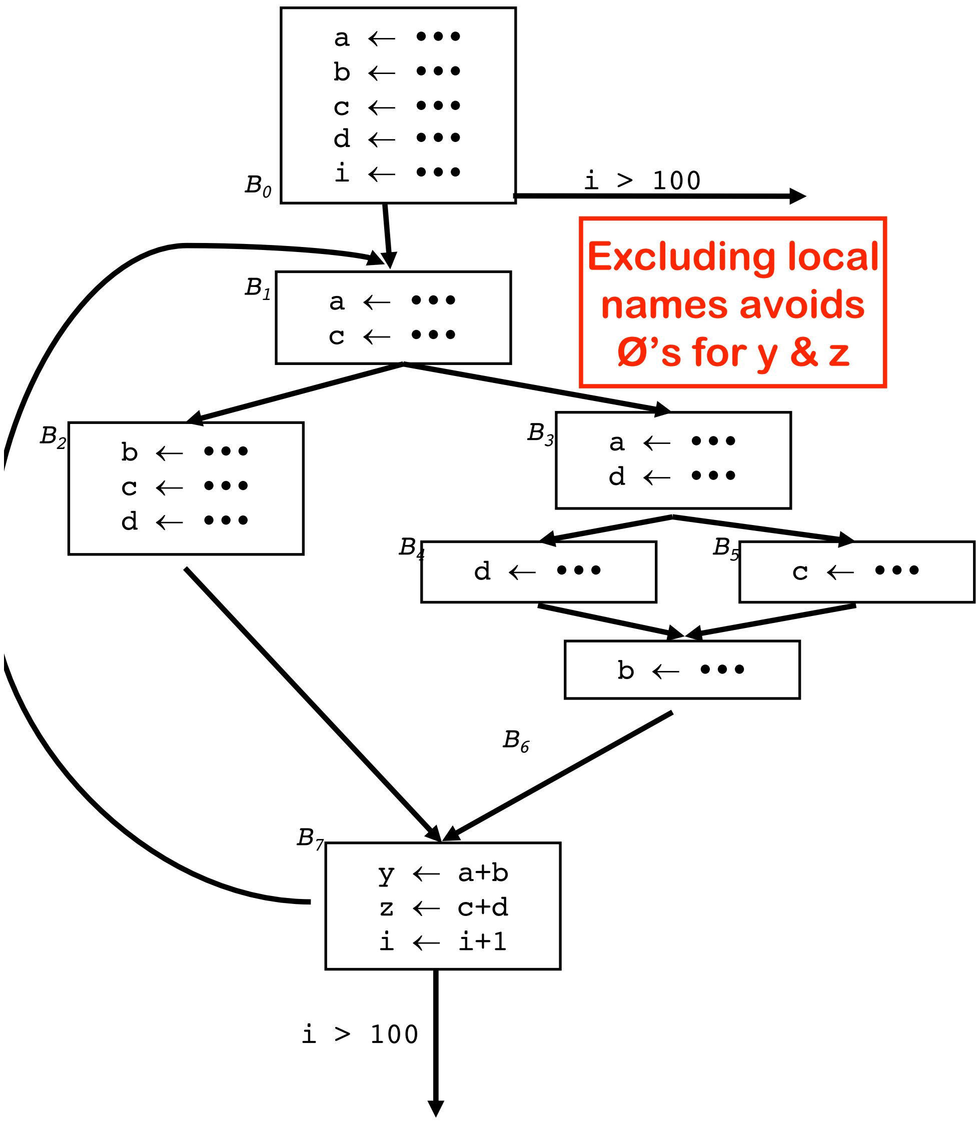
- Example

UEVAR (B7) = {a, b, c, d, i};
all others are empty set;
Globals = {a, b, c, d, i}, (y, z are local names)

- Get blocks where each of these Globals get defined

Names	a	b	c	d	i
blocks	0,1,3	0,2,6	0,1,2,5	0,2,3,4	0,7

Phase 2: inserting ϕ-functions




```
for each of the global name x
    work list = get all nodes in which x is defined
    for each node n in work list
        for each node m in DF(n)
            if (there is no ϕ-function for x in m)
                insert a ϕ-function for x to m
                work list = work list ∪ { m }
```

Names	a	b	c	d	i
blocks	0,1,3	0,2,6	0,1,2,5	0,2,3,4	0,7

Block	0	1	2	3	4	5	6	7
DF	-	1	7	7	6	6	7	1

Phase 3: renaming variables

Renaming is done by a pre-order traversal of the dominator tree, as follows:

for each node b in the dominator tree

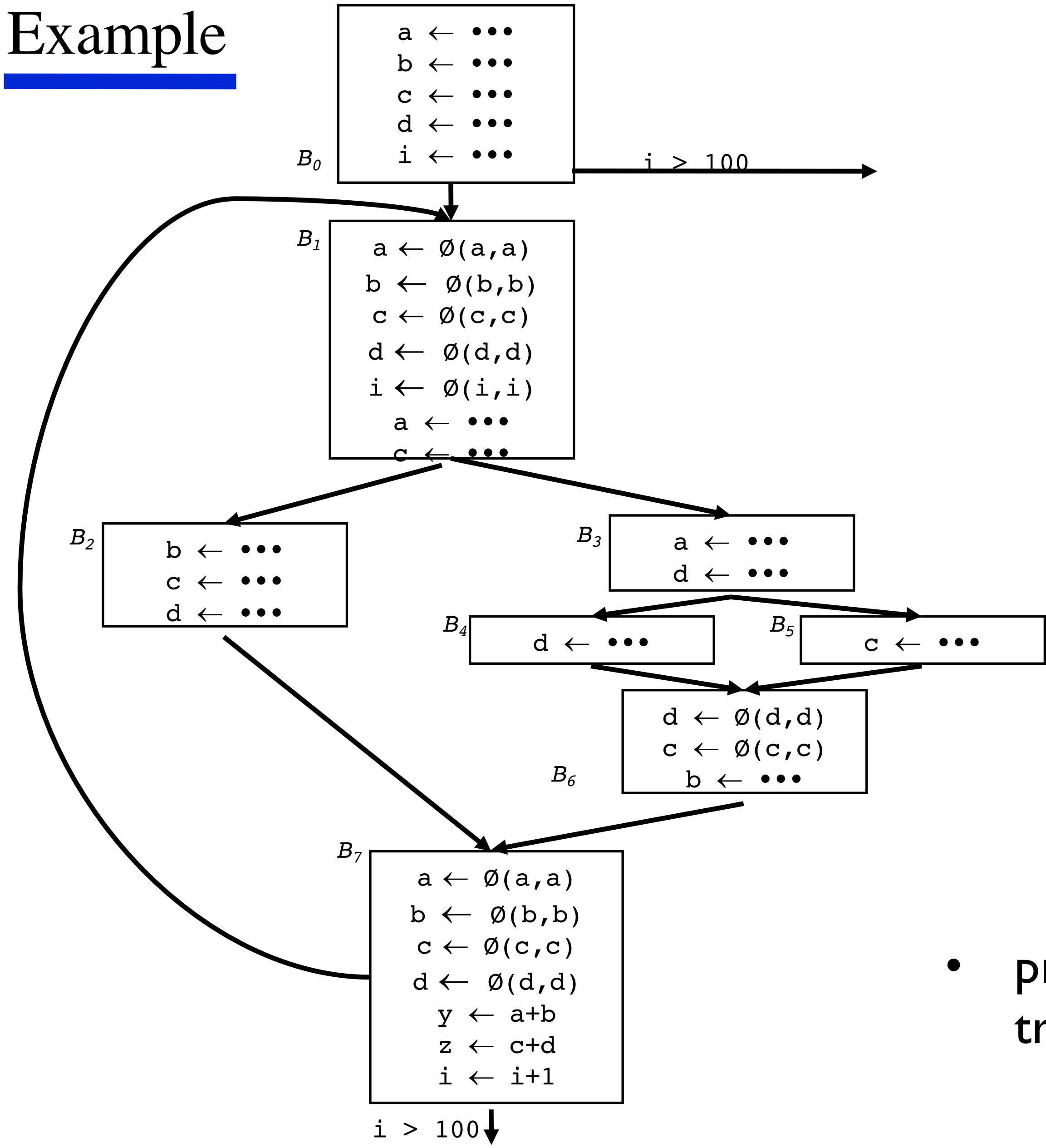
1. rename definitions and uses of variables in b
2. rename φ-functions parameters corresponding to b in all successors of n in the CFG.

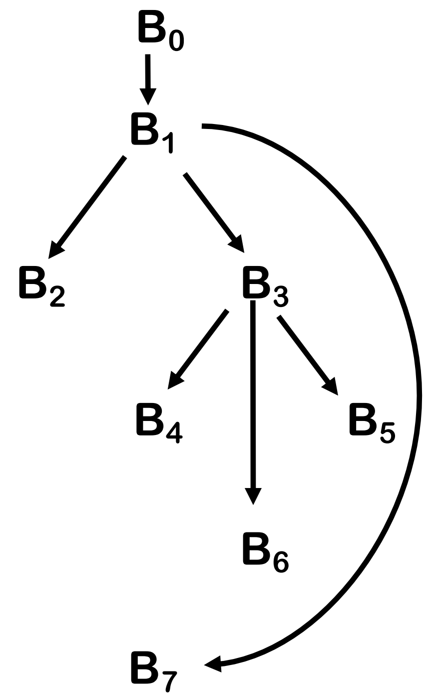
One possible Implementation via a set of stacks and counters.

1. Get the root node n_0 of the CFG
2. Call Rename(n_0)

```
Rename(b)
  for each Ø-function in b, x ¬ Ø(…)
      rename x as NewName(x)

  for each operation "x ¬ y op z" in b
      rewrite y as top(stack[y])
      rewrite z as top(stack[z])
      rewrite x as NewName(x)

  for each successor of b in the CFG
      rewrite appropriate Ø parameters

  for each successor s of b in dom. tree
      Rename(s)

  for each operation "x ¬ y op z" in b
      pop(stack[x])
```

Example

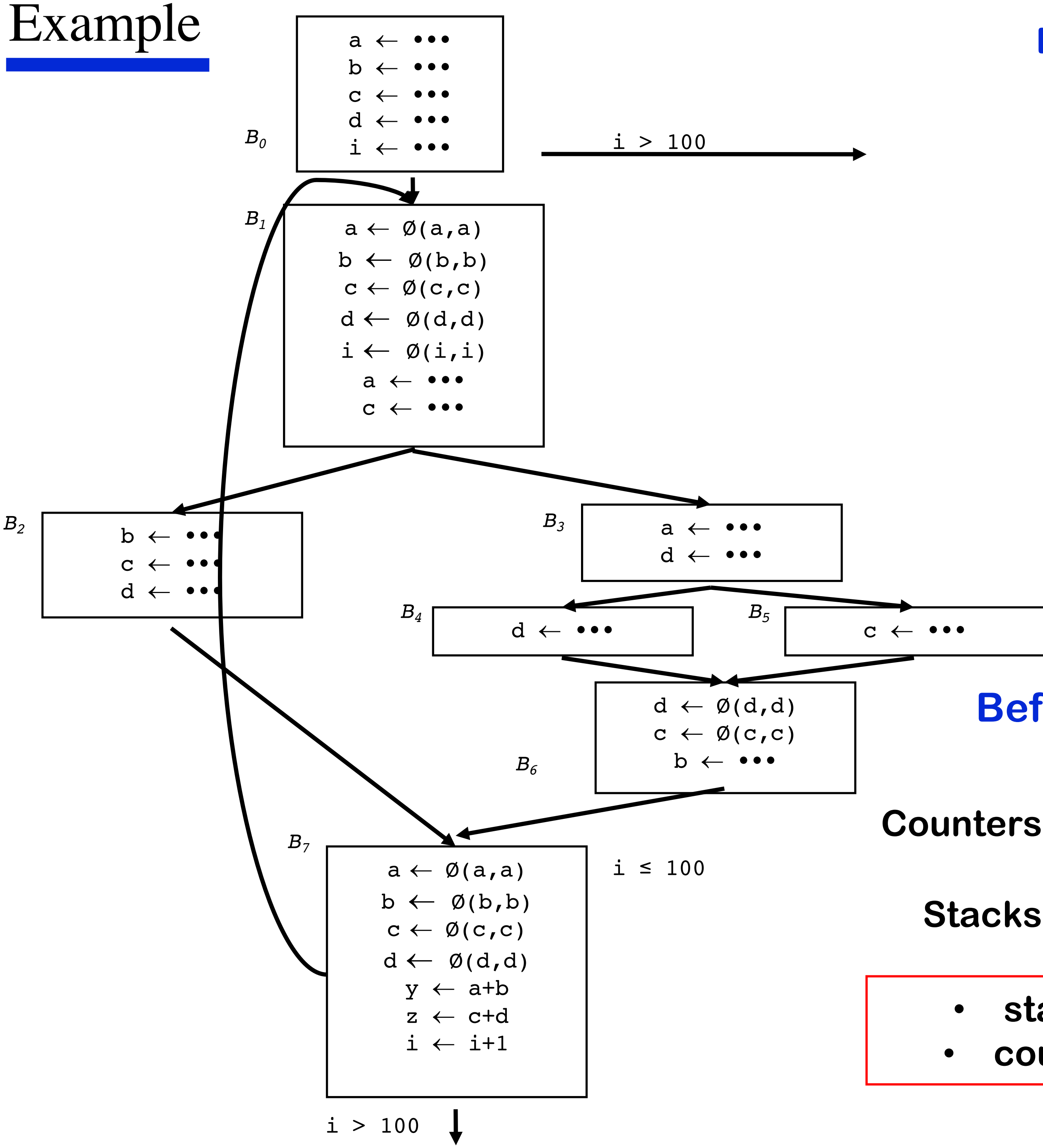




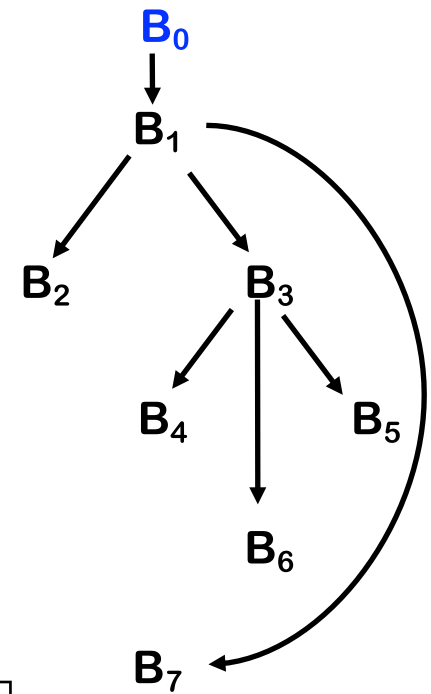
Dominance Tree

- pre-order traversal of the dominator tree: {0,1,2,3,4,5,6,7}

Example



Dominance Tree



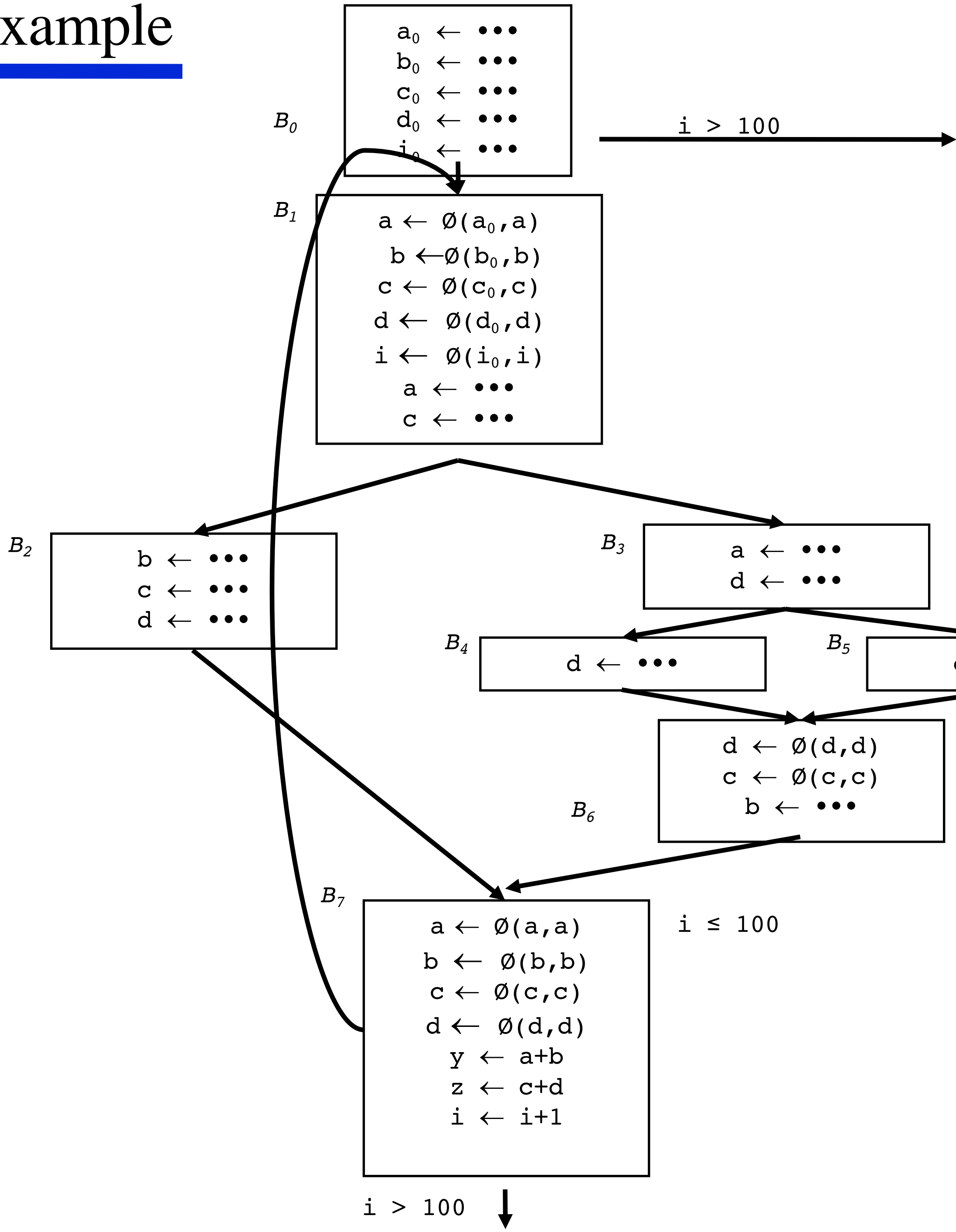
Before processing B_0

	a	b	c	d	i
Counters	0	0	0	0	0

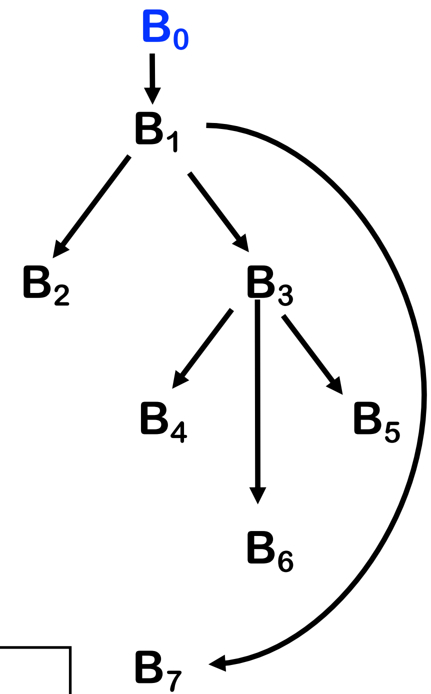
Stacks

- stacks for uses of variables
- counters for new definitions

Example



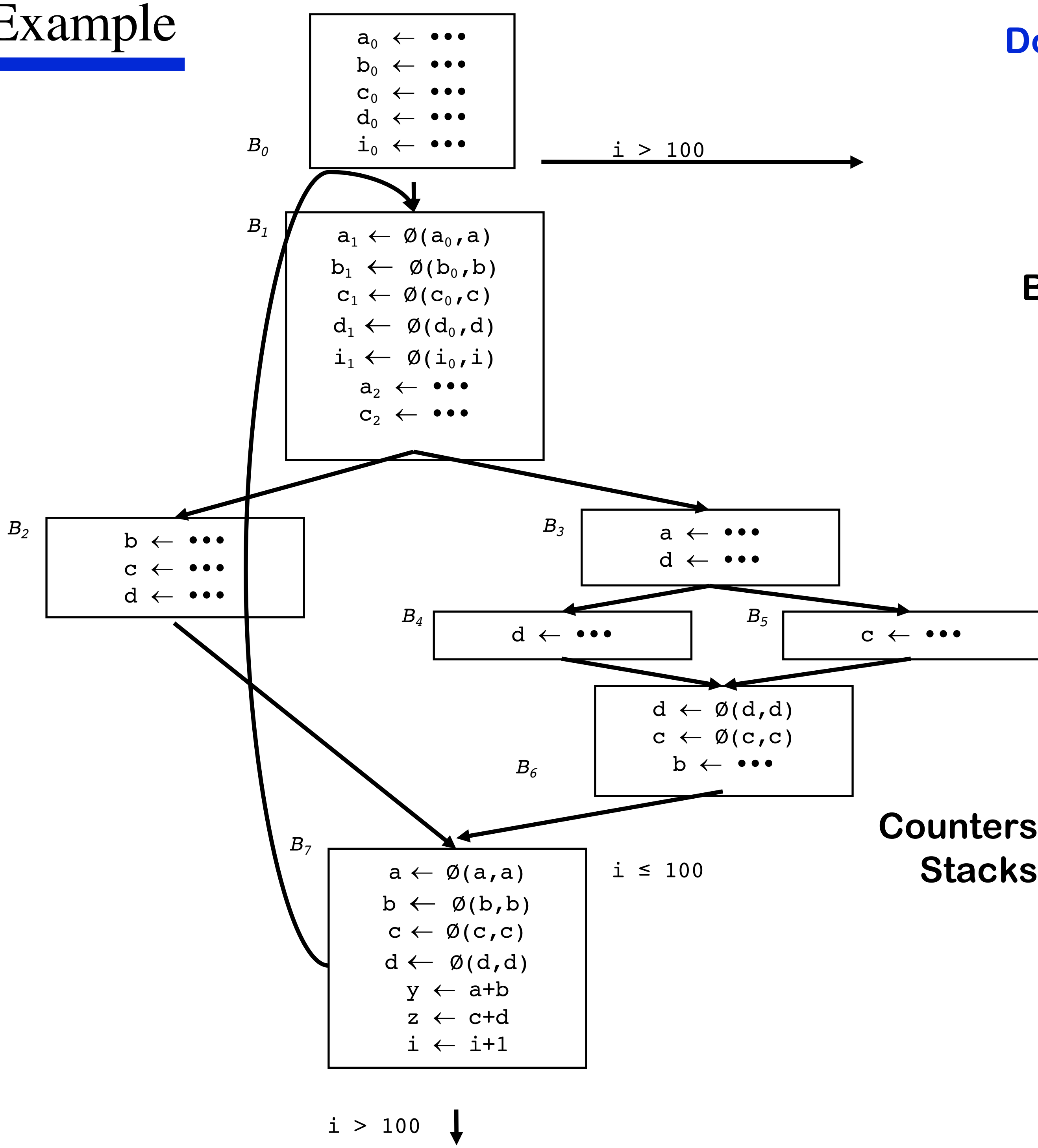
Dominance Tree



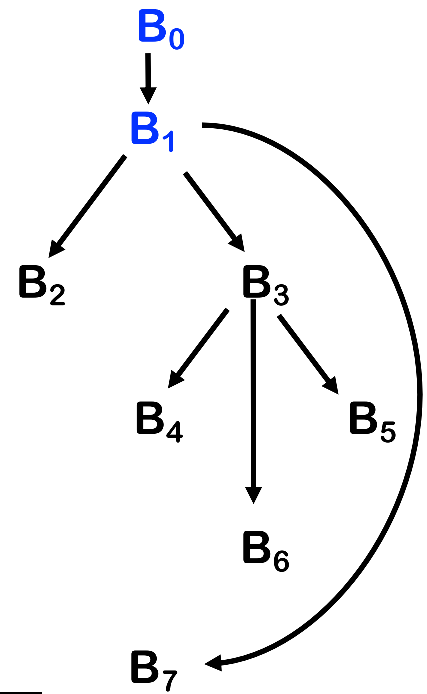
End of B_0

	a	b	c	d	i
Counters	1	1	1	1	1
Stacks	a_0	b_0	c_0	d_0	i_0

Example



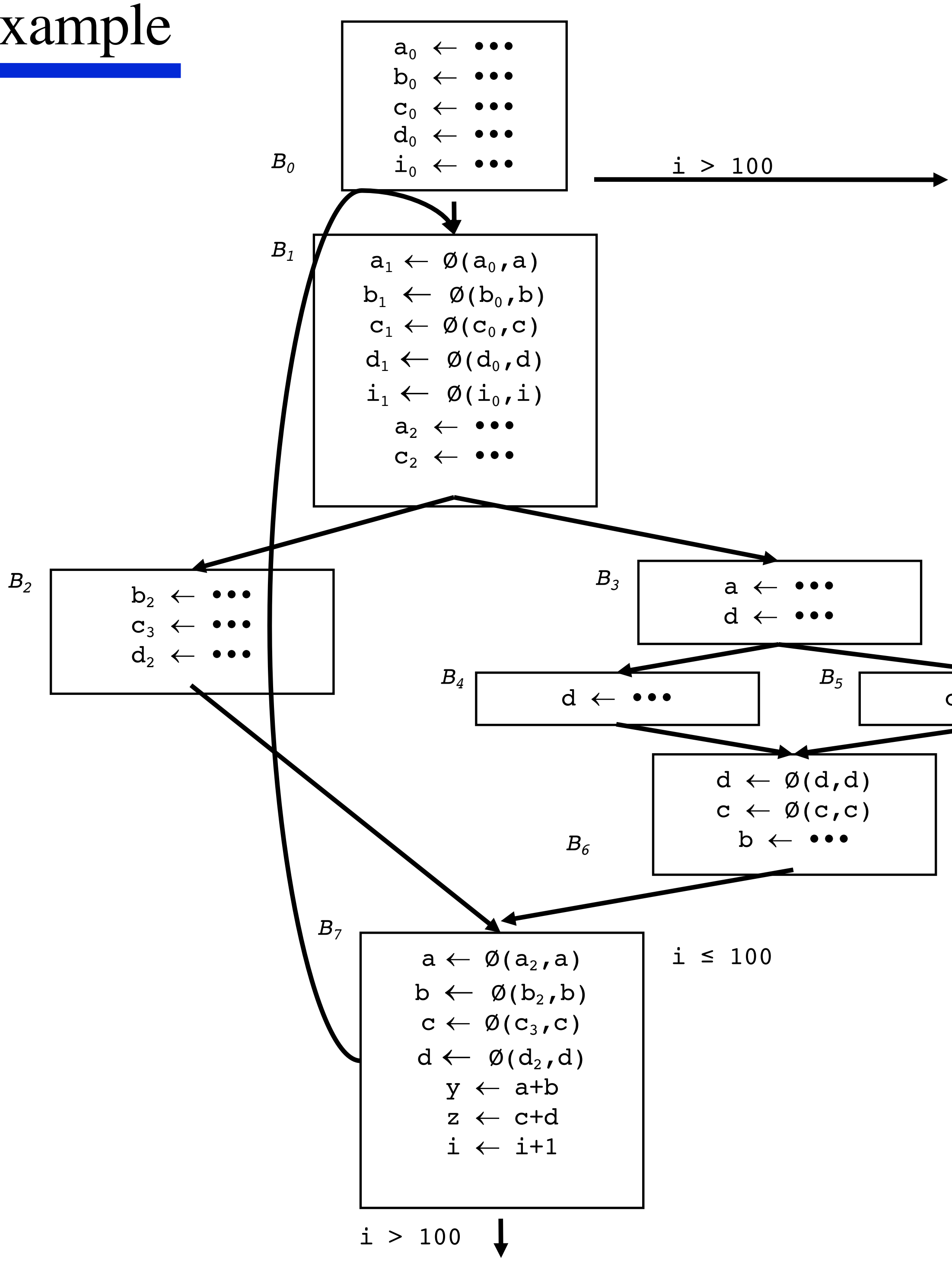
Dominance Tree



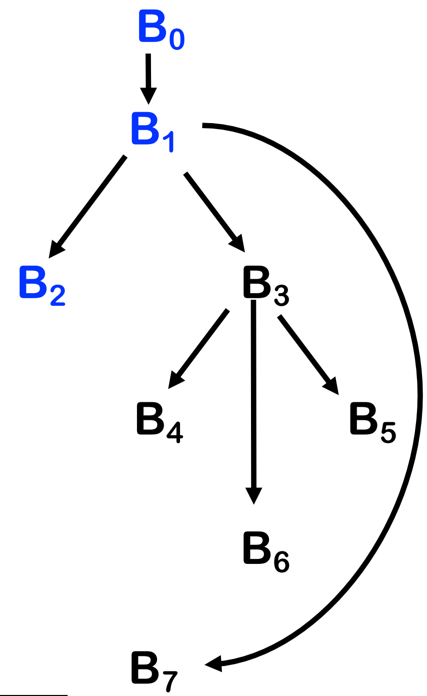
End of B_1

	a	b	c	d	i
Counters	3	2	3	2	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

Example



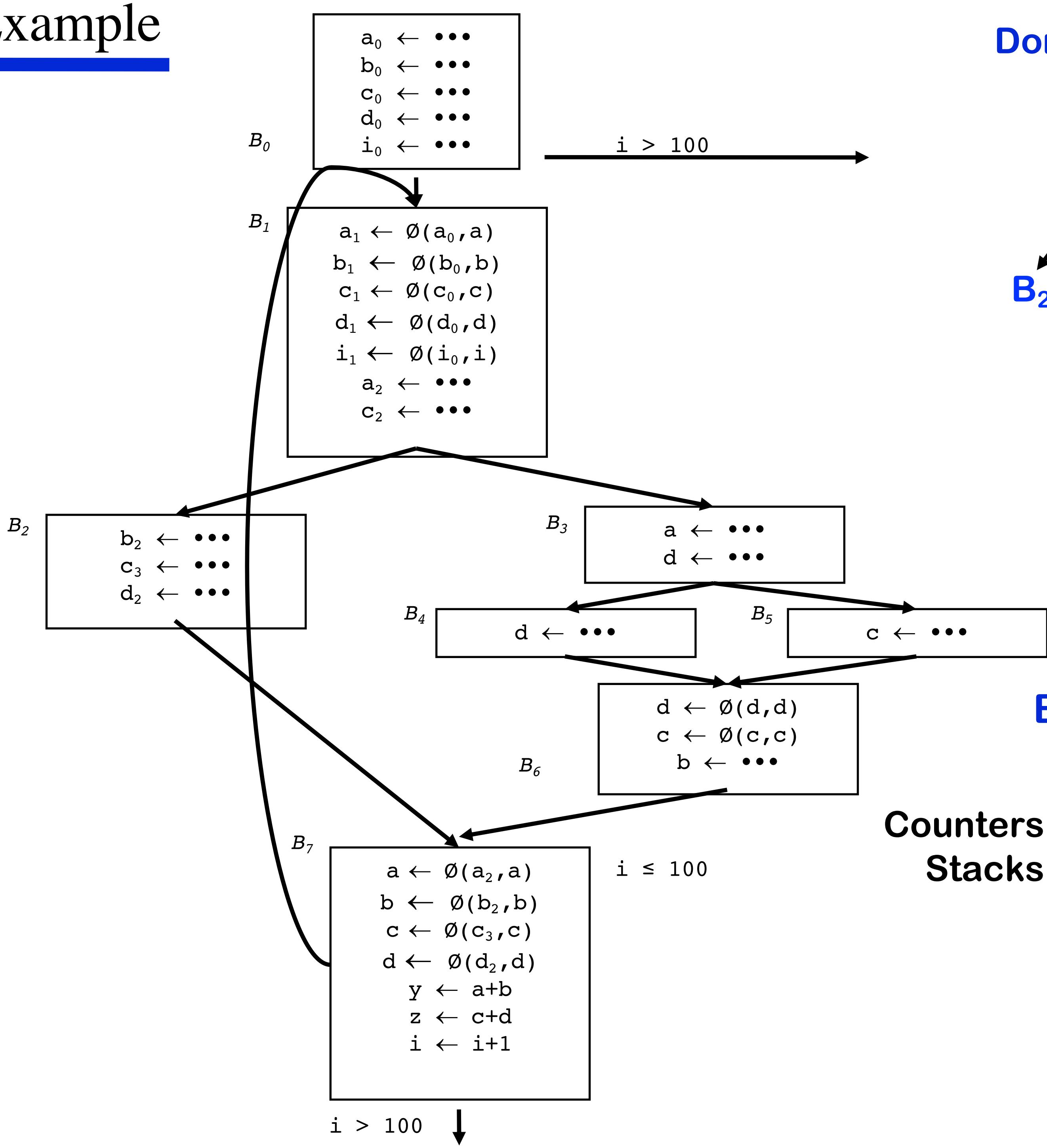
Dominance Tree



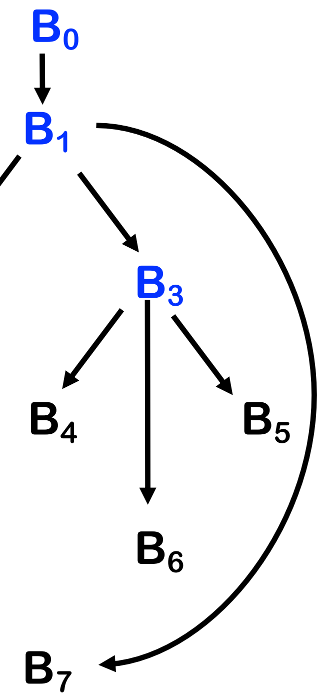
End of B_2

	a	b	c	d	i
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_2	c_2	d_2	
			c_3		

Example



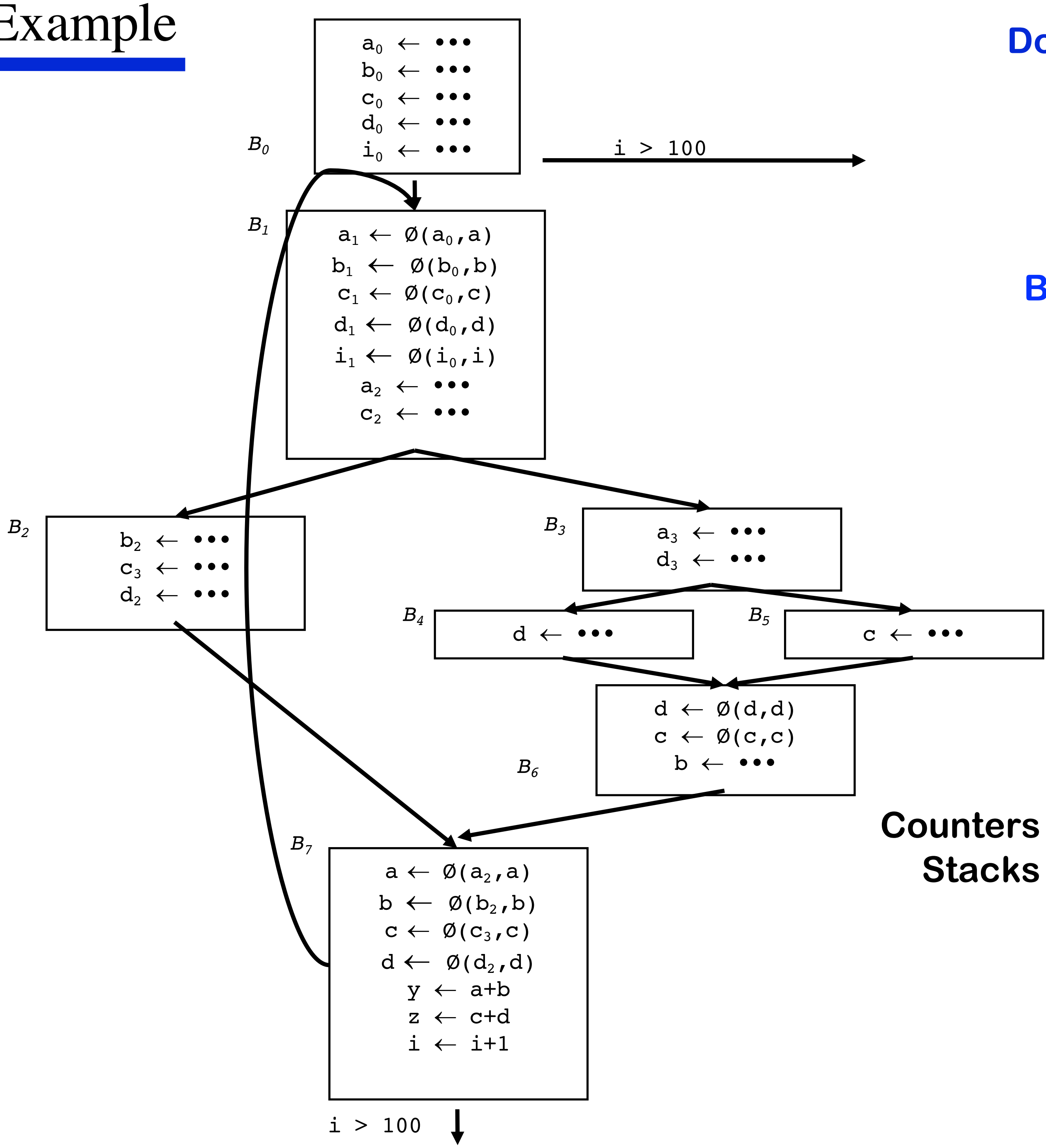
Dominance Tree

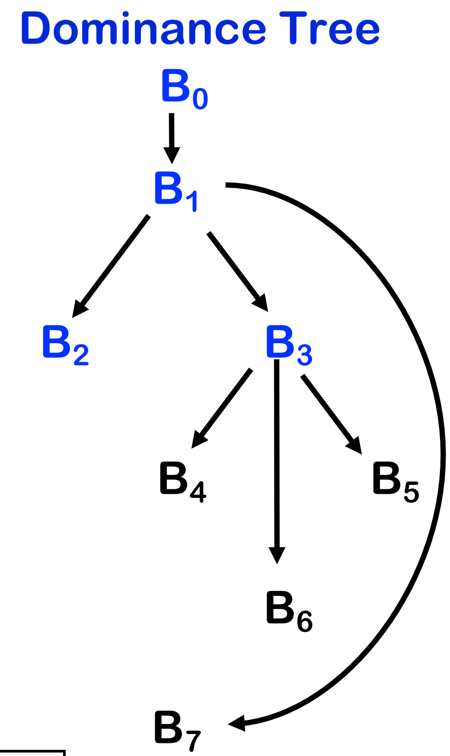


Before starting B_3

	a	b	c	d	i
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

Example

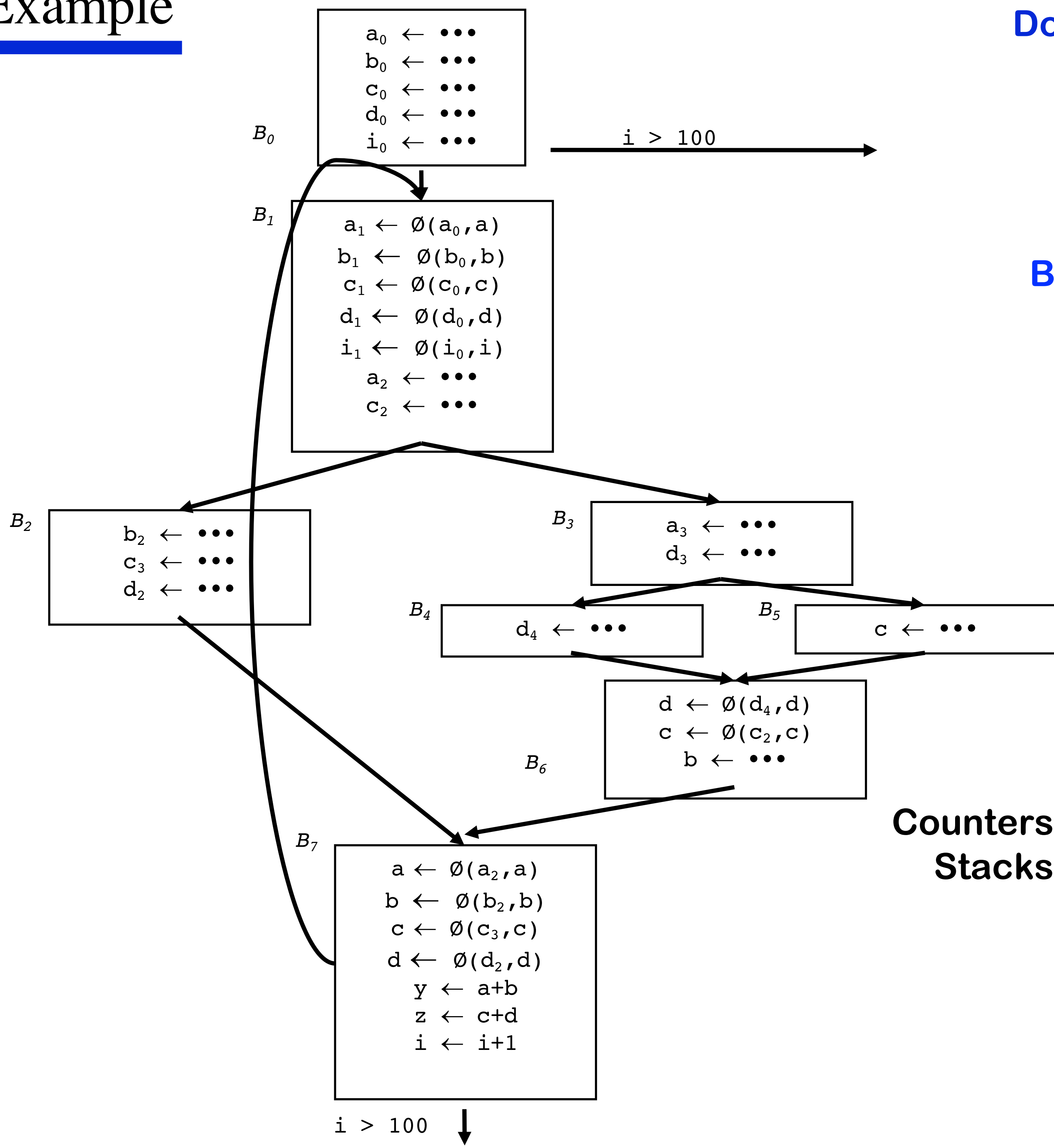




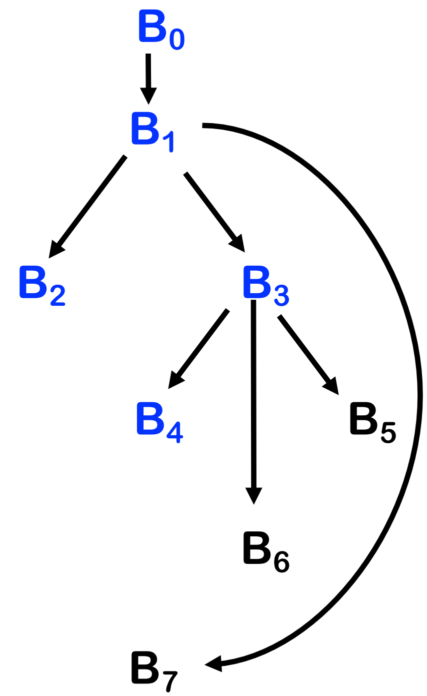
End of B_3

	a	b	c	d	i
Counters	4	3	4	4	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3				

Example



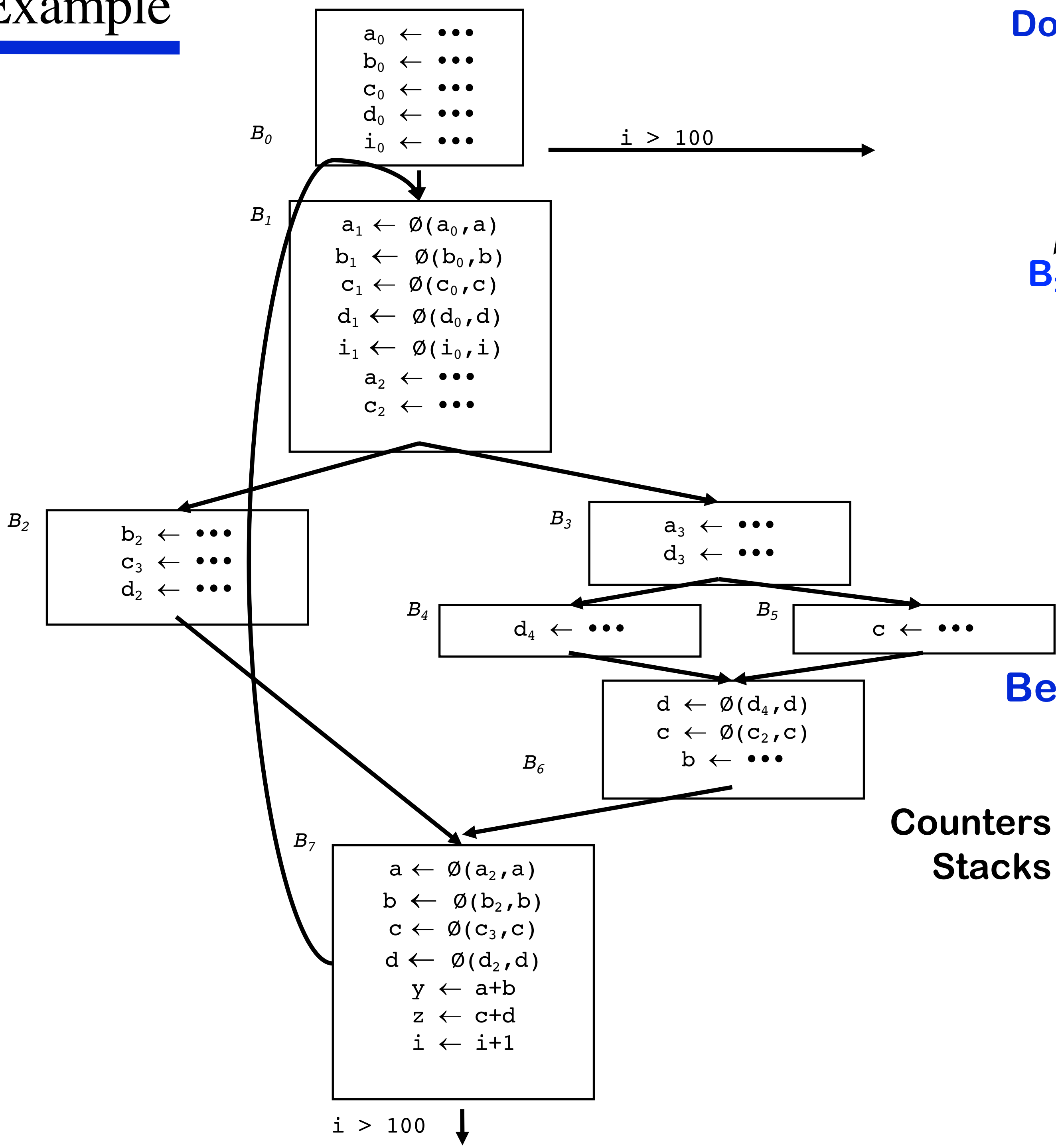
Dominance Tree

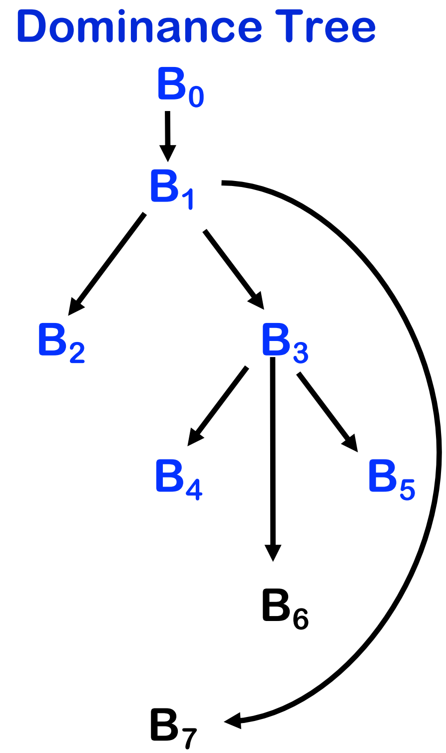


End of B_4

	a	b	c	d	i
Counters	4	3	4	5	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3			d_4	

Example

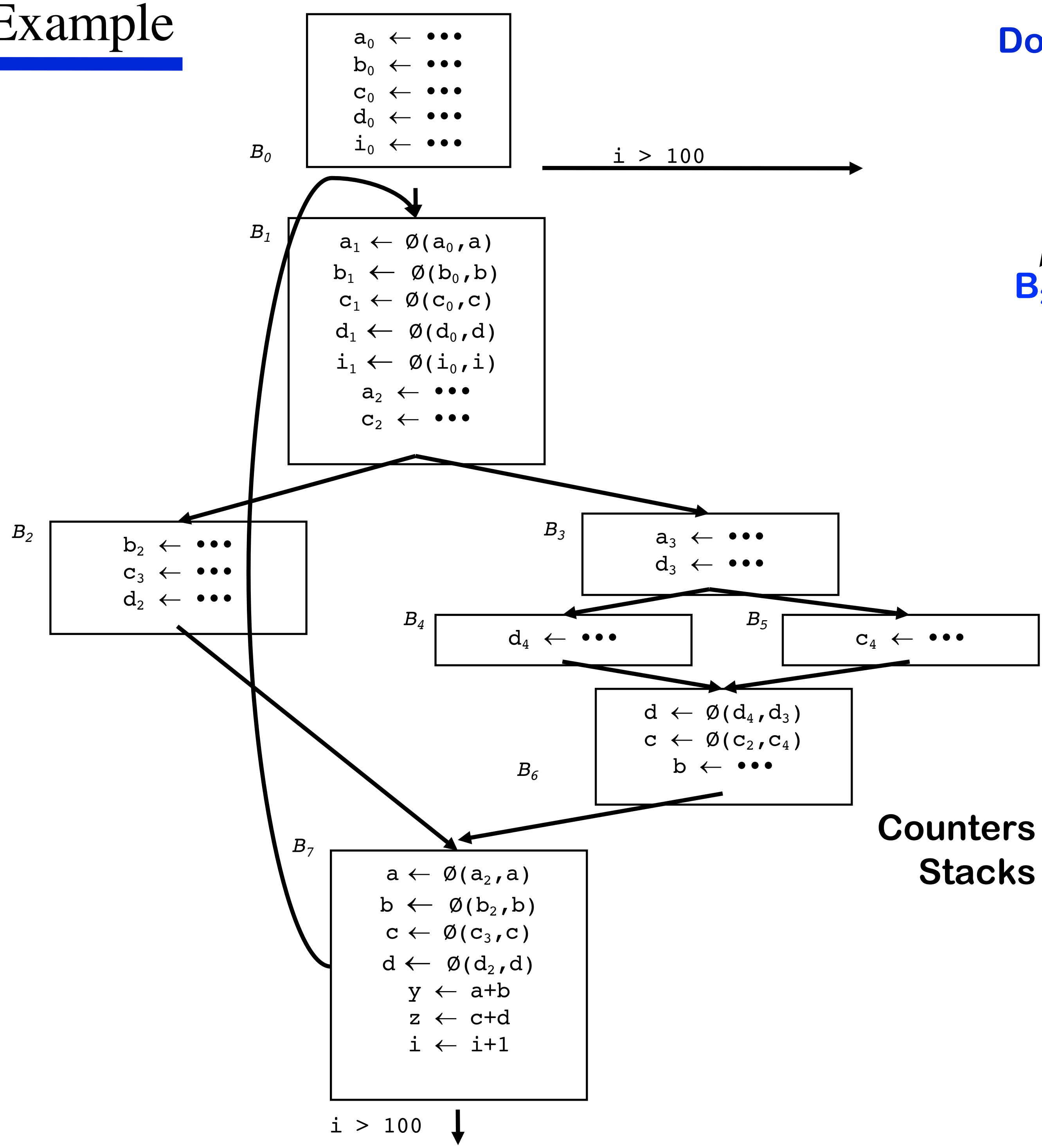




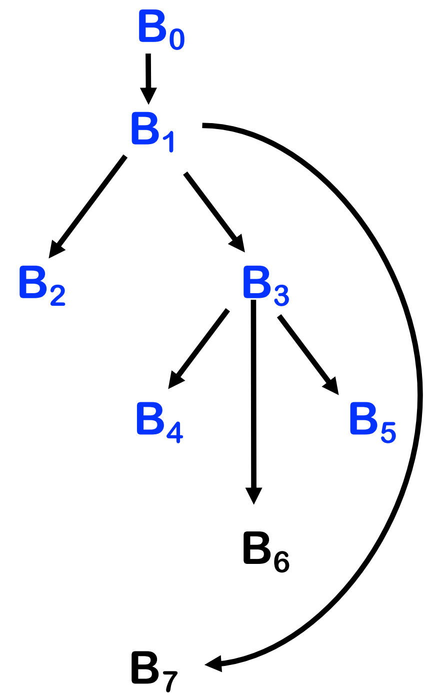
Before processing B_5

	a	b	c	d	i
Counters	4	3	4	5	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3				

Example



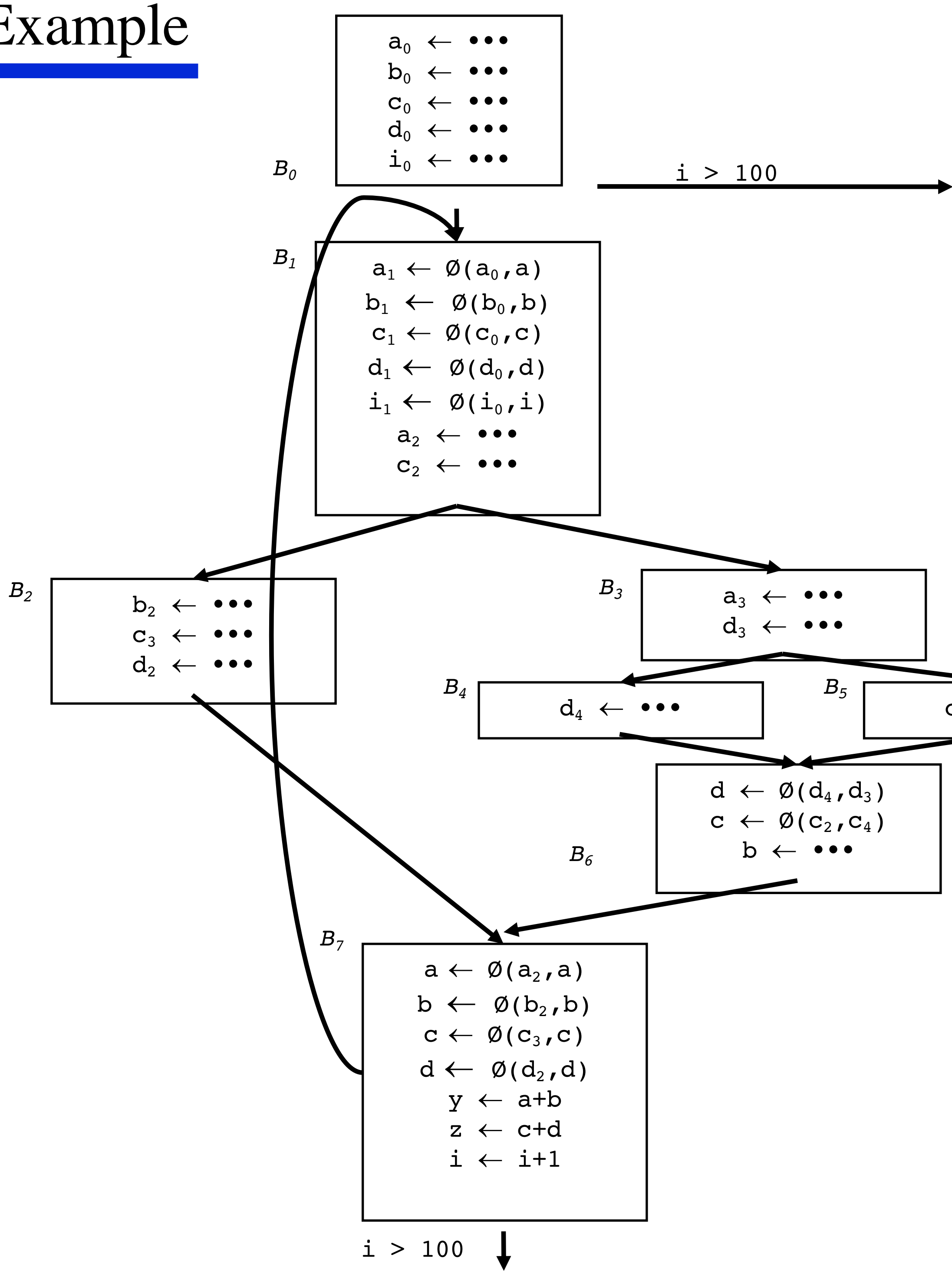
Dominance Tree



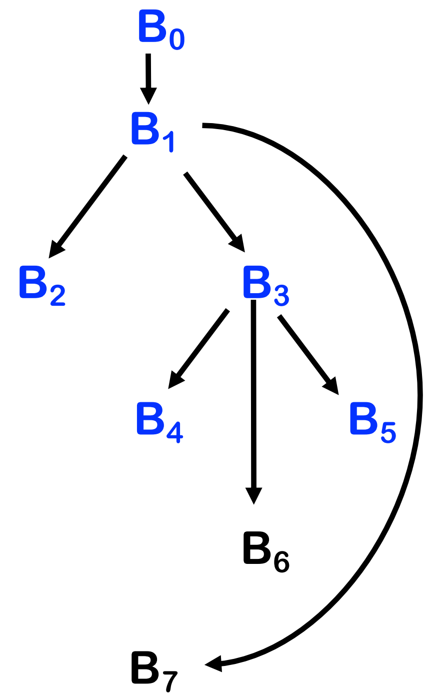
End of B_5

	a	b	c	d	i
Counters	4	3	5	5	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3		c_4		

Example



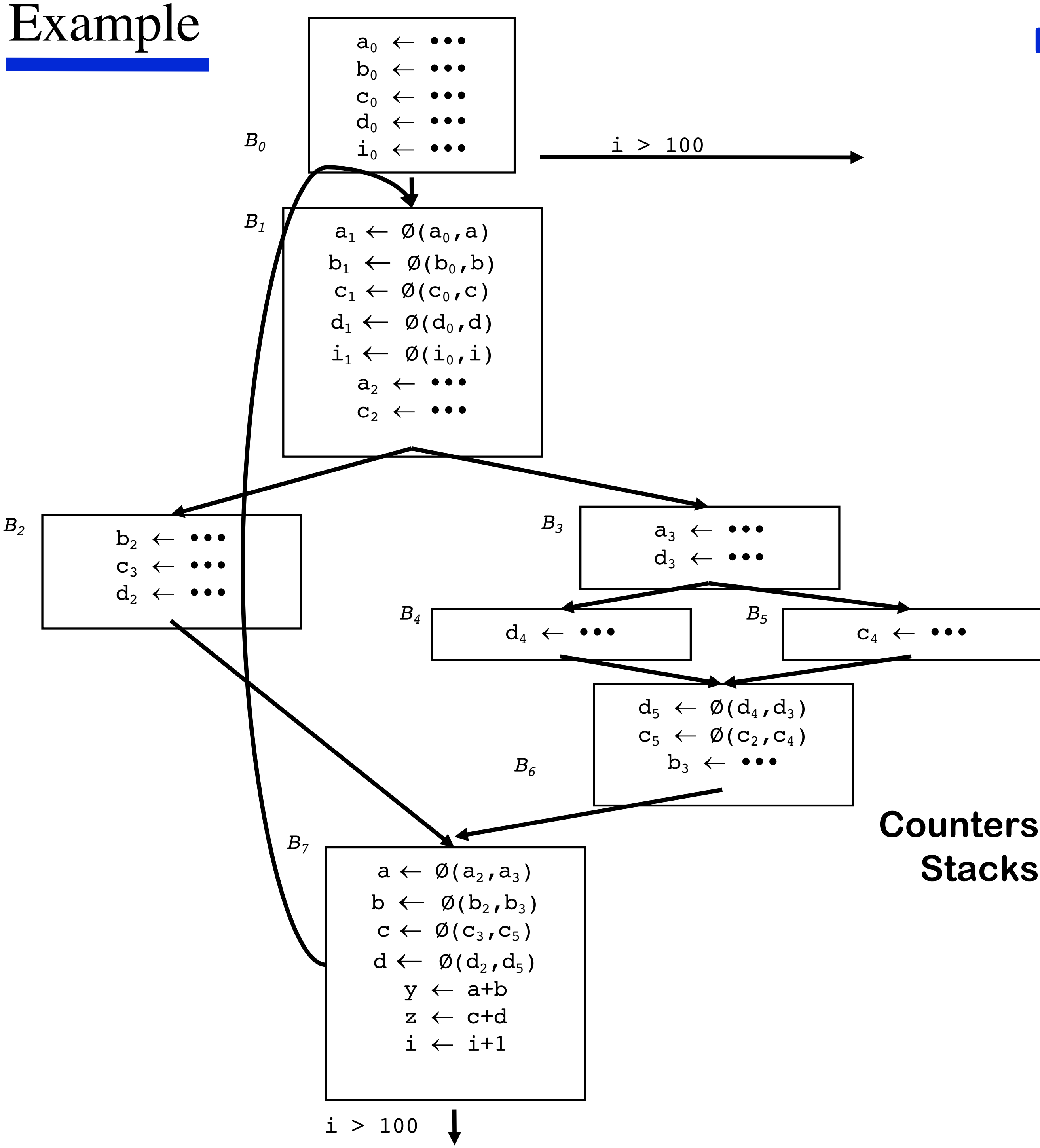
Dominance Tree



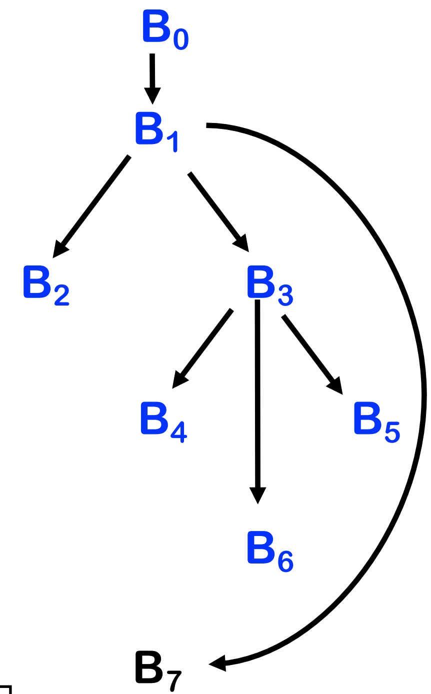
Before B_6

	a	b	c	d	i
Counters	4	3	5	5	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3				

Example



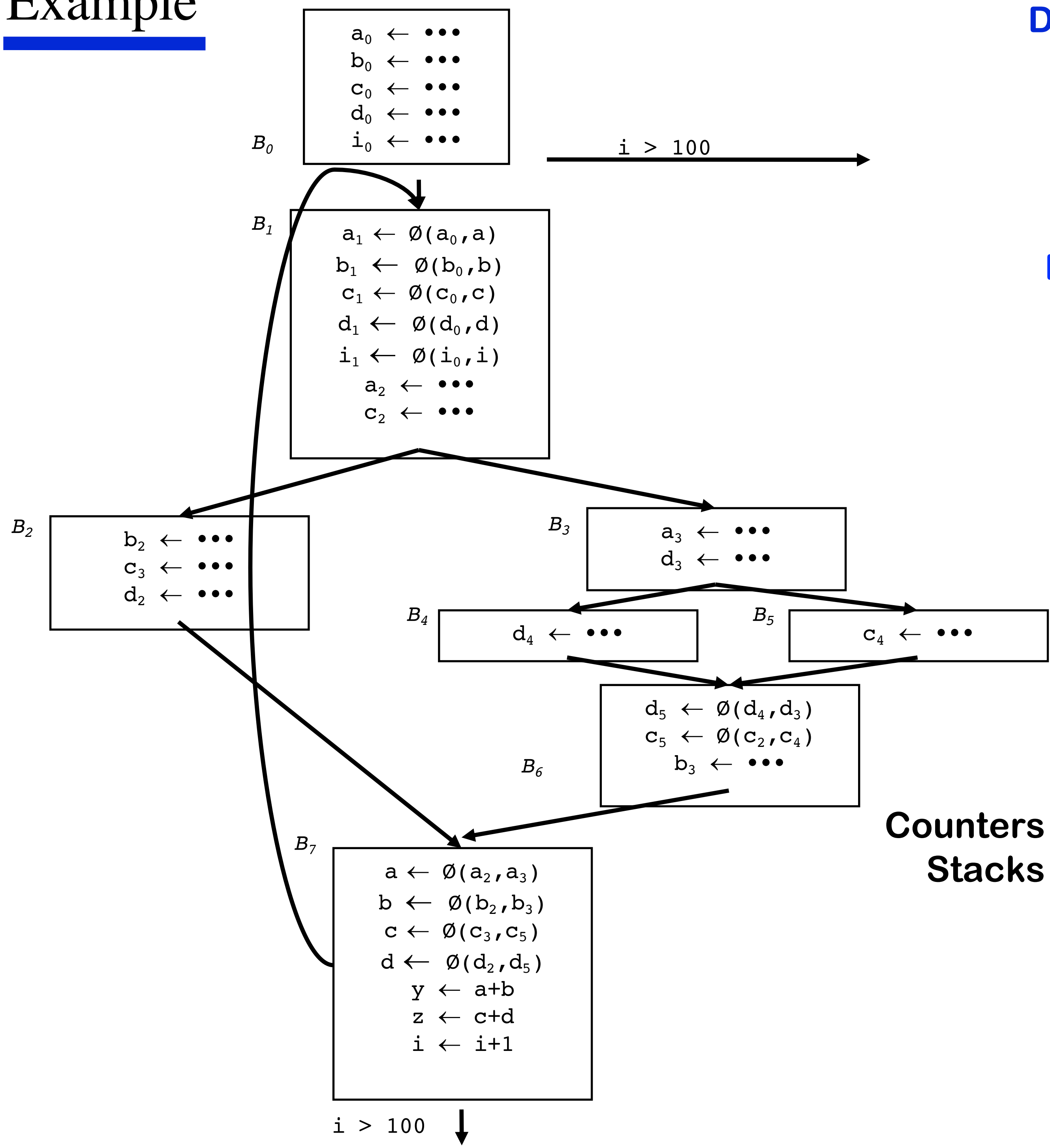
Dominance Tree



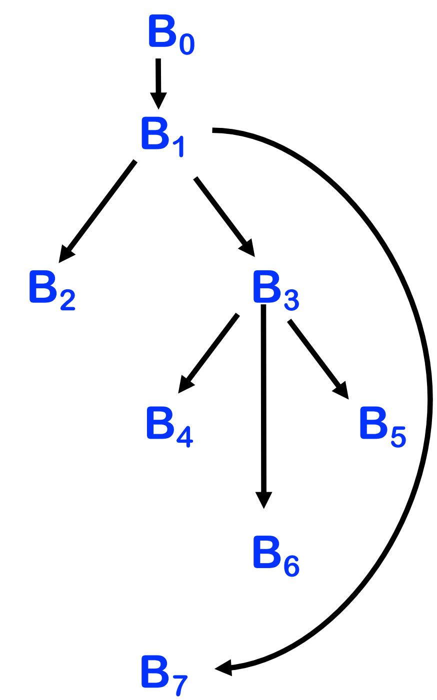
End of B_6

	a	b	c	d	i
Counters	4	4	6	6	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_3	c_2	d_3	
	a_3		c_5	d_5	

Example



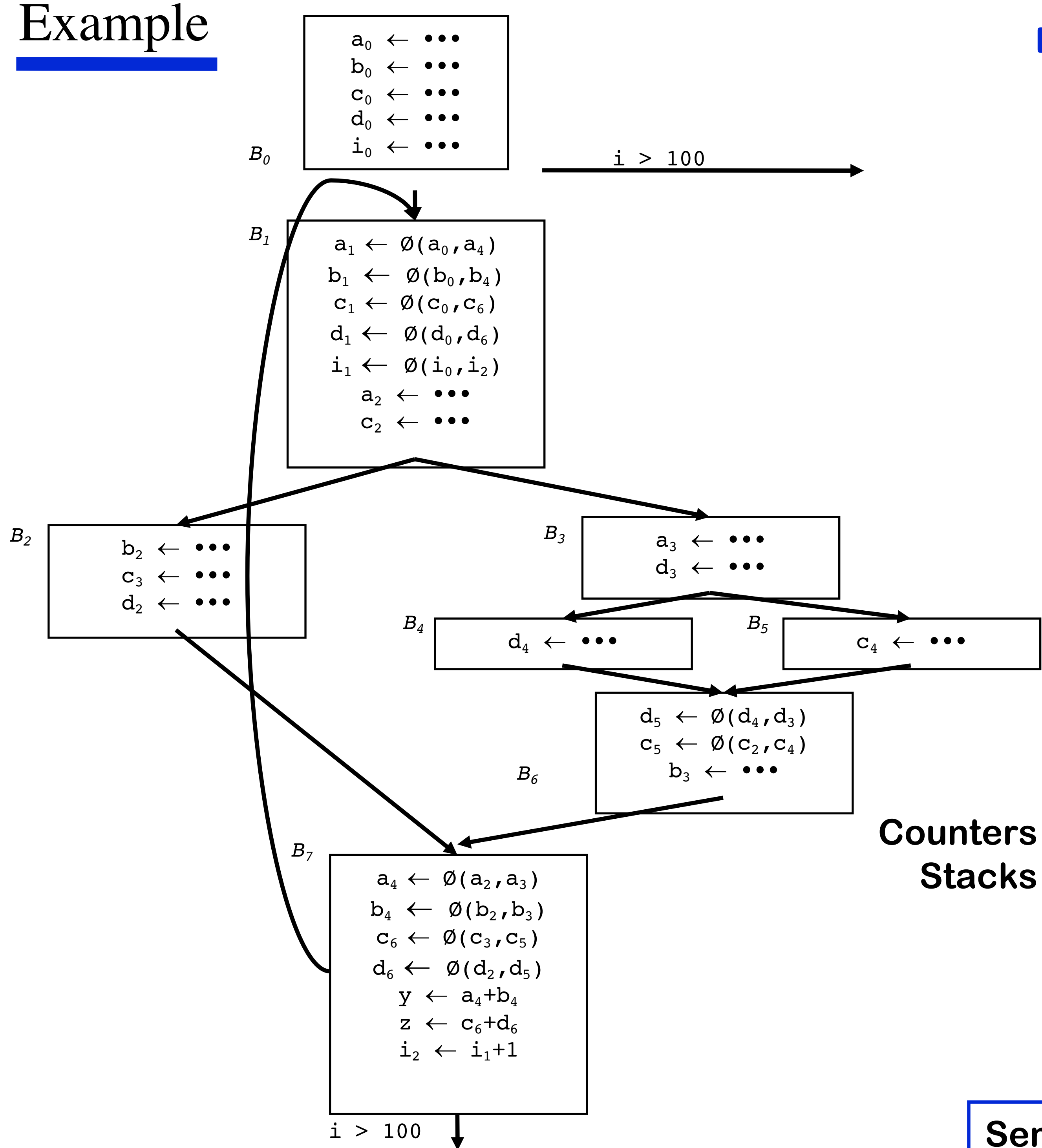
Dominance Tree



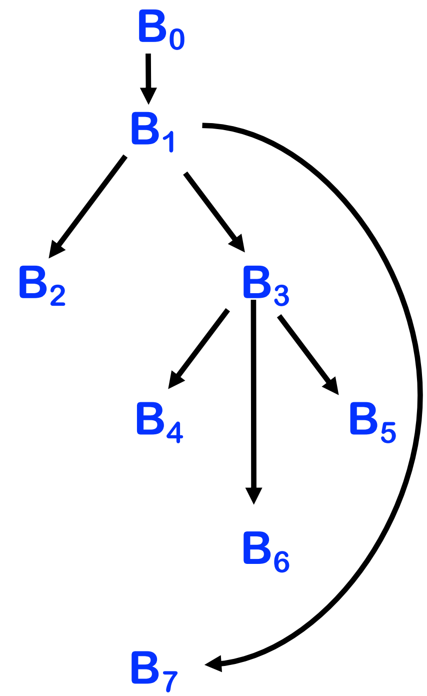
Before B_7

	a	b	c	d	i
Counters	4	4	6	6	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

Example



Dominance Tree



End of B_7

	a	b	c	d	i
Counters	5	5	7	7	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_4	c_2	d_6	i_2
	a_4		c_6		

Semi-pruned SSA, done!

Semi-pruned SSA V.S. Pruned SSA

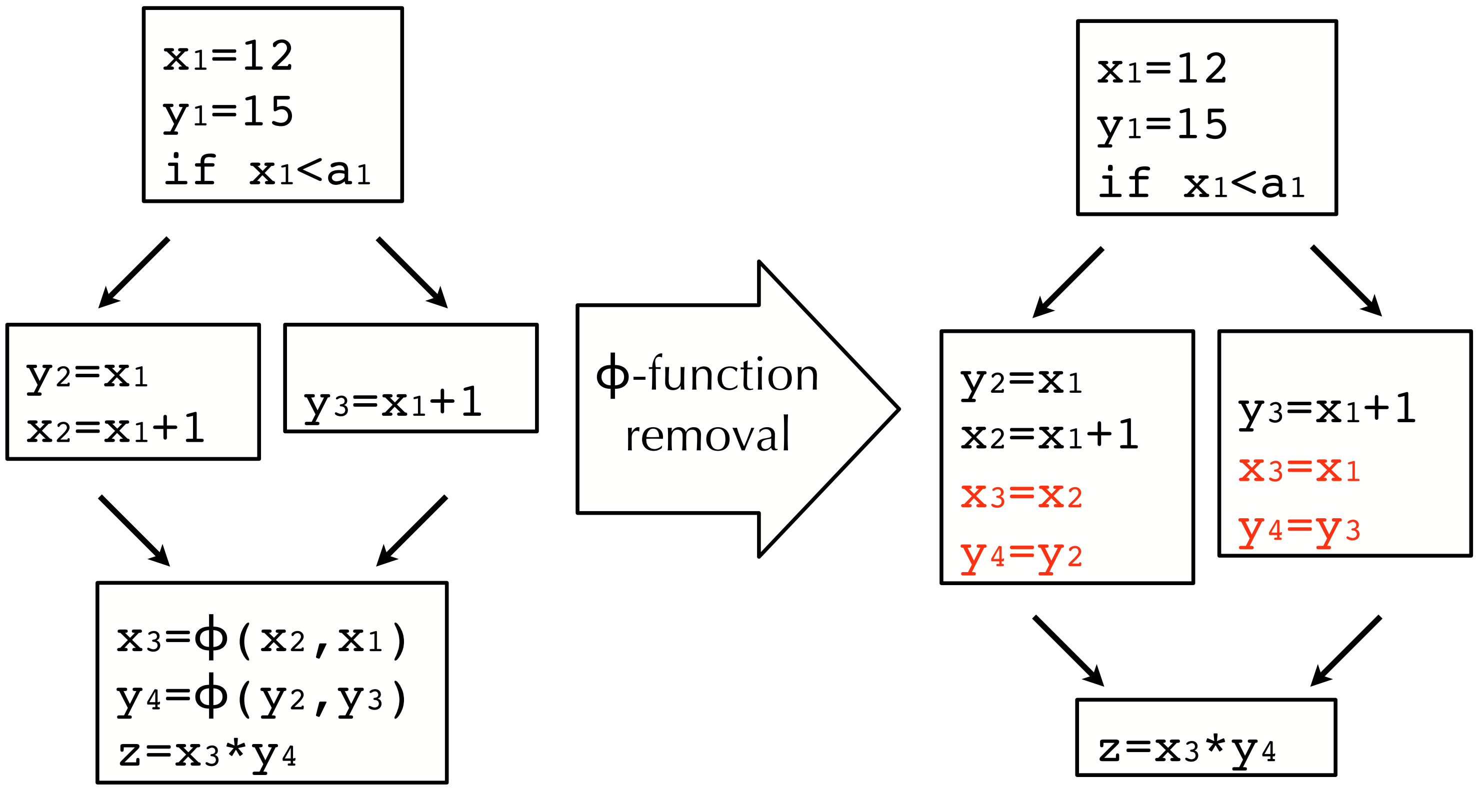
- Semi-pruned SSA: discard names used in only one block
  - Significant reduction in total number of Ø-functions
  - Needs only local Live (appearance) information (cheap to compute)

- Pruned SSA: only insert Ø-functions where their value is live
  - Inserts even fewer Ø-functions, but costs more to do

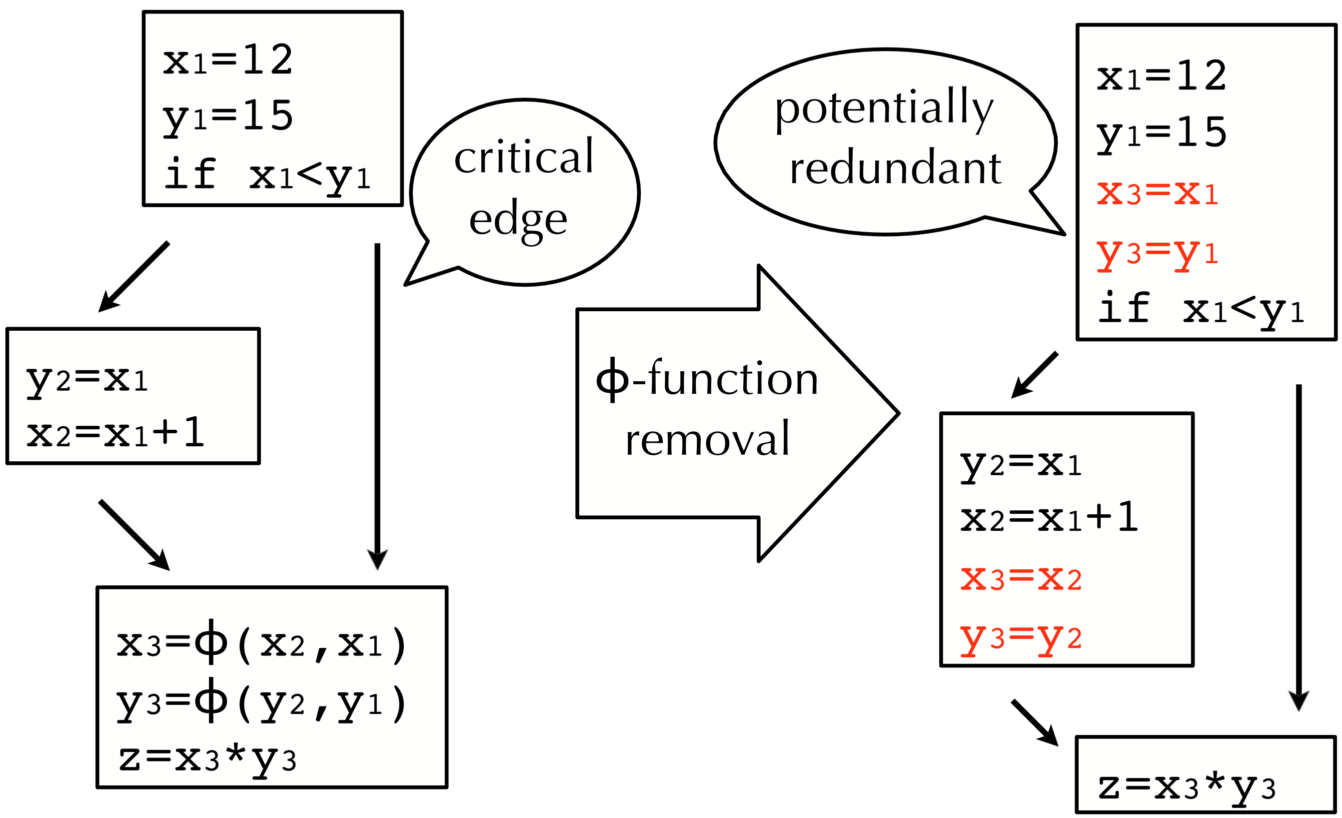
Removing ϕ-functions

- After the program has been turned into SSA form and the various optimizations performed on that representation, it must be transformed into executable form.
- This implies in particular that φ-functions must be removed, as they cannot be implemented on standard machines.

Removing ϕ-functions



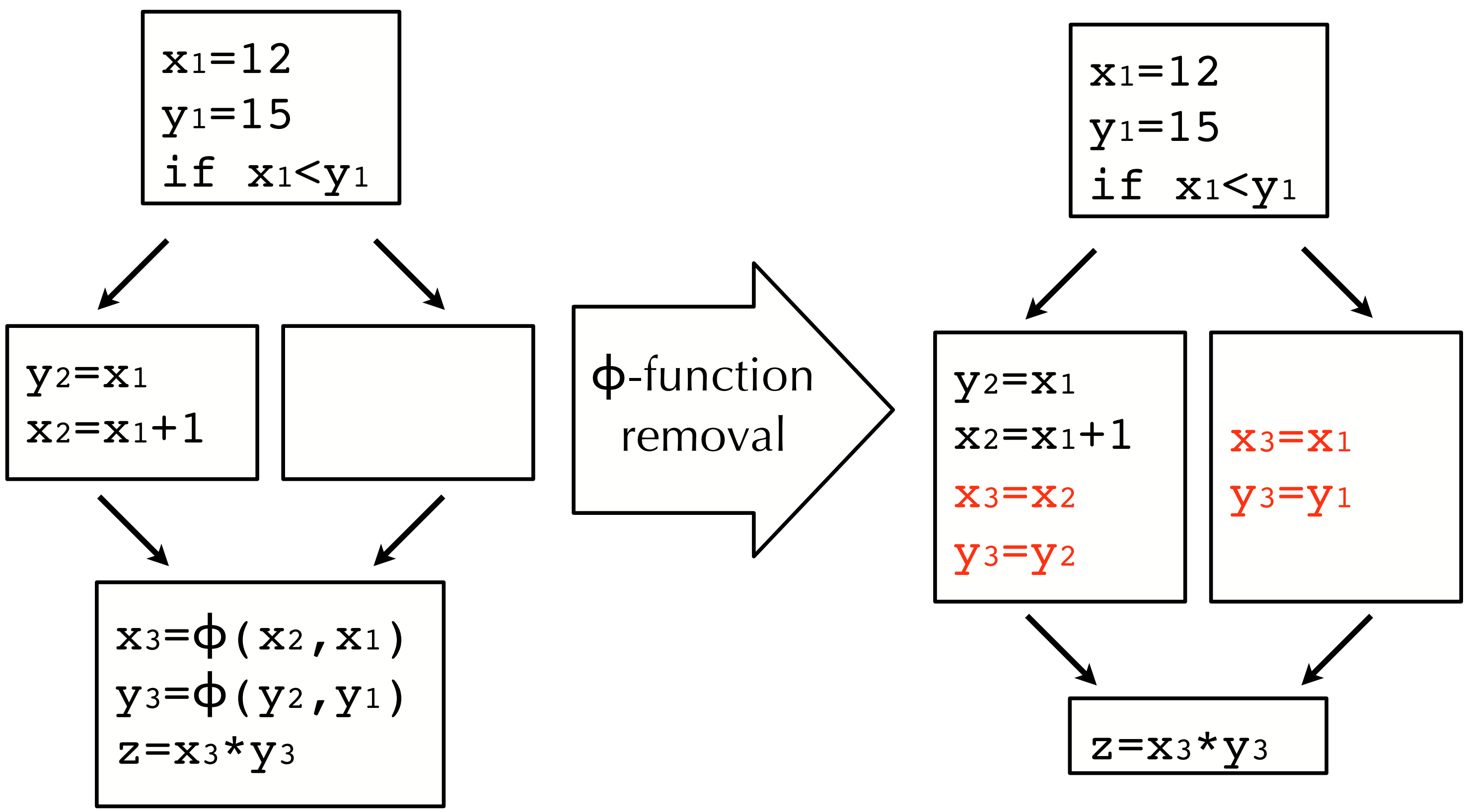
Potential redundancy with critical edge



Critical edges

- CFG edges that go from a node with multiple successors to a node with multiple predecessors are called critical edges.
- While removing φ-functions, the presence of a critical edge from n_1 to n_2 leads to the insertion of redundant *move instructions* in n_1, corresponding to the φ-functions of n_2.
- Ideally, they should be executed only if control reaches n_2 later, but this is not certain when n_1 executes.

With edge splitting



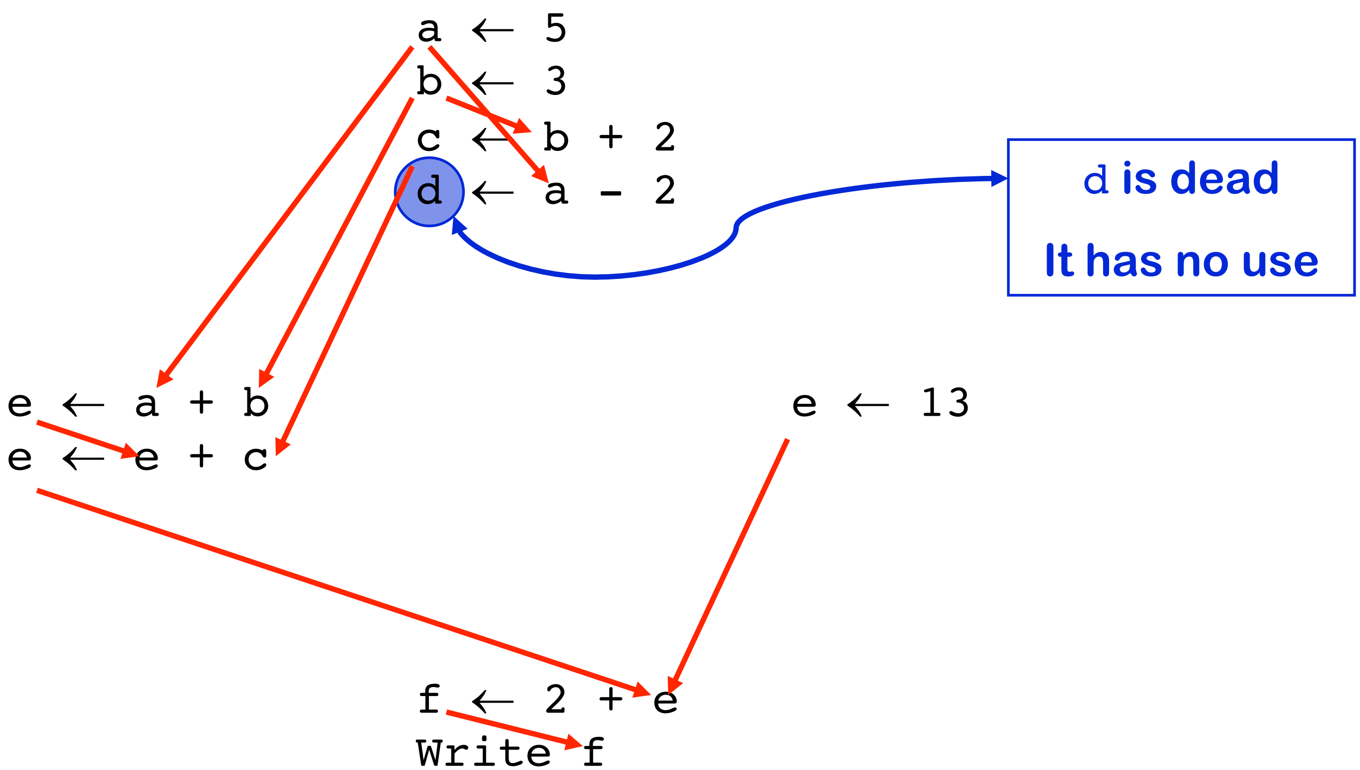
Dead Code Elimination

- Useful statements
  - Output statements (e.g., printf)
  - Statements that compute values used by useful statements

- Algorithm to eliminate dead code
  - Start with absolutely useful statements
  - Repeatedly adds statements that compute variables used in current useful statements
    - through def-use chains (reaching definitions)

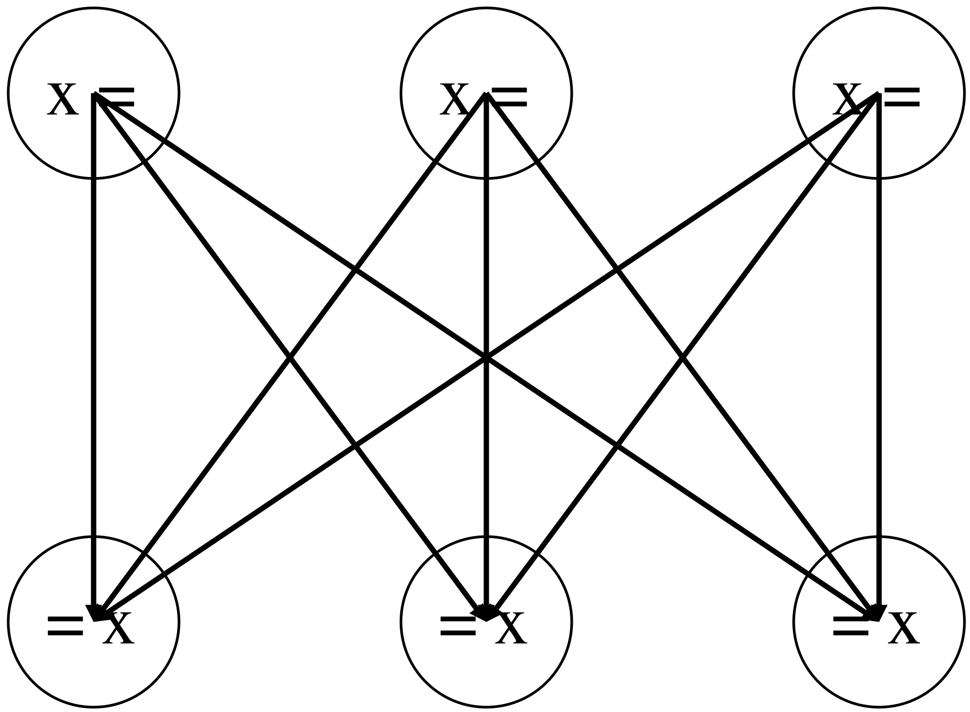
Dead-code Elimination

- Using def-use chain (review):



Def-use w/o SSA form

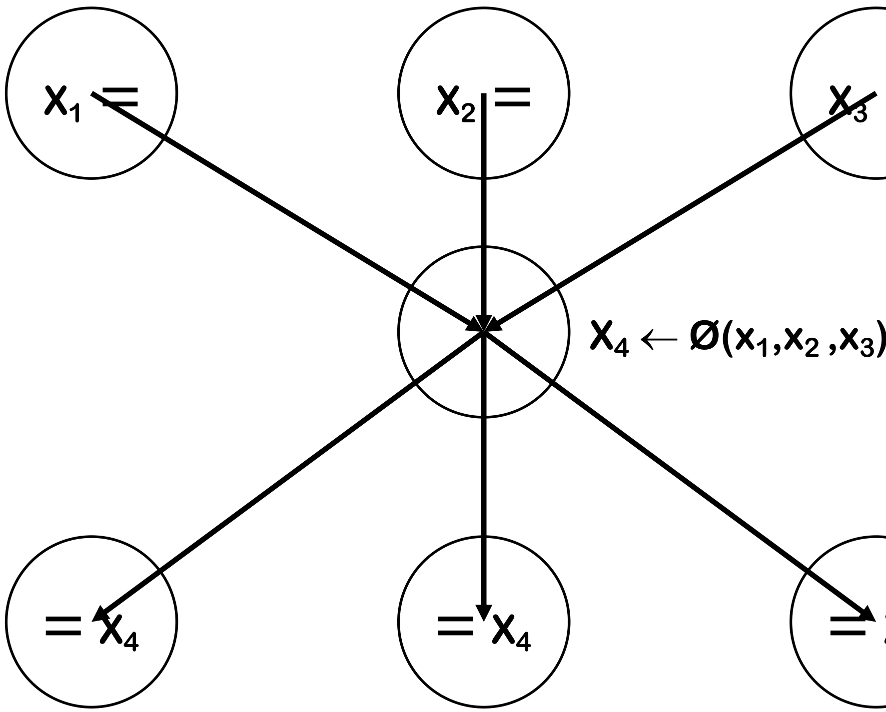
- Def-use edges grow very large



caused by
branches

Def-use with SSA Form

- Edges reduced from 9 to 6



Example

```
if (x > 0) {
        printf("greater than zero");
}
```

- The printf statement (I/O statement) is inherently live. You also need to mark the "if (x>0)" live because the 'print' statement is control dependent on the 'if'.

Post-dominator Relation

- If X appears on every path from START to Y, then X dominates Y.
- If X appears on every path from Y to END, then X postdominates Y.
- Postdominator Tree
  - END is the root
  - Any node Y other than END has ipdom(Y) as its parent
  - Parent, child, ancestor, descendant

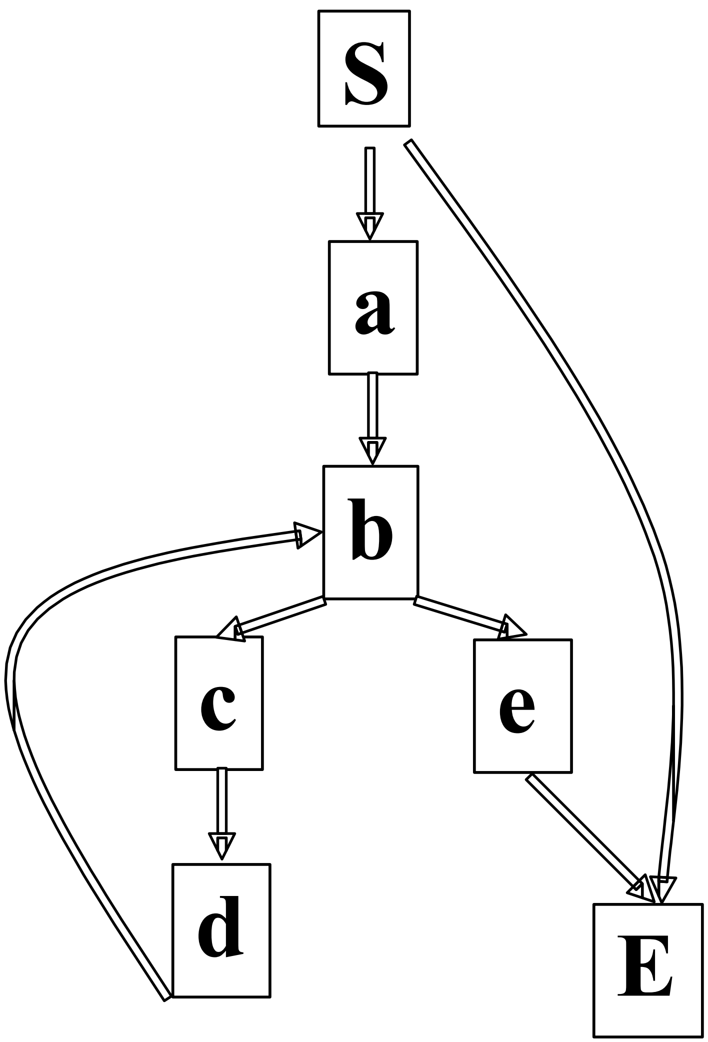
Control Dependence

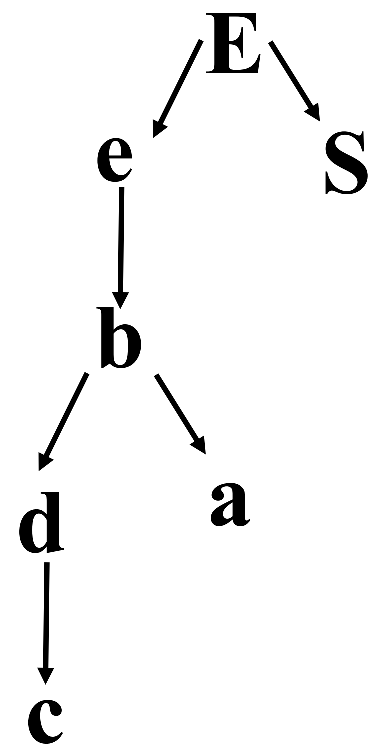
- There are two possible definitions.

- Node w is control dependent on edge (u→v) if
  - w postdominates v
  - If w ≠ u, w does not postdominate u

- Node w is control dependent on node u if there exists an edge u→v
  - w postdominates v
  - If w ≠ u, w does not postdominate u

Example





Pdom Tree

Control Dep Relation

	a	b	c	d
S->a	√	√		
b->c		√	√	√

Control Dependence V.S. Dominator Frontier

- Reverse control flow graph (RCFG)

- Let X and Y be nodes in CFG. X in DF(Y) in CFG iff Y is control dependent on X in RCFG.

- DF(Y) in CFG = conds(Y) in RCFG, where conds(Y) is the set of nodes that Y is control dependent on.

Using SSA for Dead Code Elimination

```
Mark
  for each op i
     clear i's mark
     if i is critical then
        mark i
        add i to WorkList

  while (Worklist ≠ Ø)
     remove i from WorkList
         (i has form "x←y op z" )
     if def(y) is not marked then
        mark def(y)
        add def(y) to WorkList
     if def(z) is not marked then
        mark def(z)
        add def(z) to WorkList

     for each b ∈ RDF(block(i))
        mark the block-ending
           branch in b
        add it to WorkList
```

```
Sweep
  for each op i
     if i is not marked then
        if i is a branch then
           rewrite with a jump to
              i's nearest useful
              post-dominator
        if i is not a jump then
           delete i
```

Notes:

- Eliminates some branches
- Reconnects dead branches to the remaining live code

Summary

In general, using SSA leads to

- Cleaner formulations
- Better results
- Faster algorithms

Important concepts of control dependence

- postdominator, reverse dominance frontier
- Relations between control dependence and dominance relations

Dead code elimination algorithm.