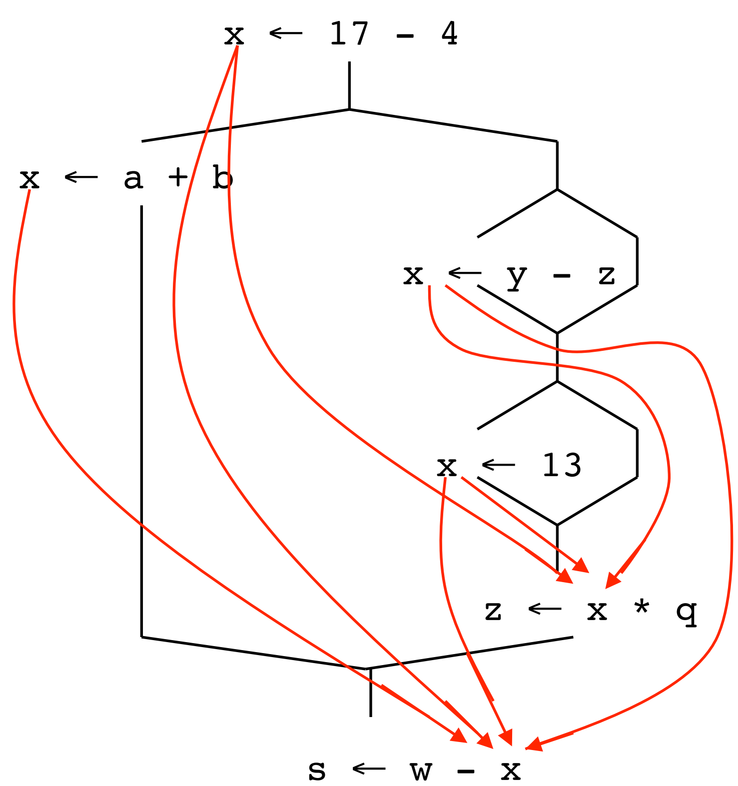
COMP 512
Rice University
Spring 2015

Construction of Static Single-Assignment Form

Citation numbers refer to entries in the EaC2e bibliography.

DEF-USE Chains (see last lecture)

Example



- Figure shows only those **DEF-USE** chains that involve *x*
- Figure ignores other variables
- Notice that multiple **DEF**s can reach a given **USE** & each **USE** can reach multiple **DEF**s
  - → Some authors call a connected set of **DEF**s & **USE**s as a "web"
  - → **DEF-USE** webs are live ranges in global register allocation [75,74]

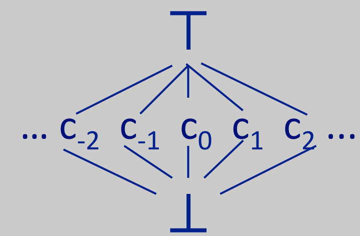
Review from prior lectures

$|CONSTANTS| = |variables|$

Constant Propagation, The Old Way

Transformation: Global Constant Folding

- Along every path to *p*, *v* has same known value
- Specialize computation at *p* based on *v*'s value



The Lattice *C*

Data-flow problem: Constant Propagation

Domain is the set of pairs $<v_i,c_i>$ where v_i is a variable and $c_i \in C$

$$CONSTANTS(b) = \wedge_{p \in preds(b)} f_p(CONSTANTS(p))$$

- $\wedge$ performs a pairwise meet on two sets of pairs
- $f_p(x)$ is a block specific function that models the effects of block p on the $<v_i,c_i>$ pairs in *x*

Form of *f* is quite different than in the other GDFAPs that we have seen

Constant propagation is a forward flow problem

Review from prior lectures

Constant Propagation, The Old Way

Meet operation requires more explanation

- $c_1 \wedge c_2 = c_1$ if $c_1 = c_2$, else $\bot$ (bottom & top as expected)

What about f_p ?

> f_p does not fit into the mold of the functions in our Kam-Ullman rapid frameworks.

- If *p* has one statement then

  $x \leftarrow y$ with $CONSTANTS(p) = \{\ldots<x,l_1>,\ldots<y,l_2>\ldots\}$

  then $f_p(CONSTANTS(p)) = CONSTANTS(p) - <x,l_1> + <x,l_2>$

  $x \leftarrow y\ op\ z$ with $CONSTANTS(p) = \{\ldots<x,l_1>,\ldots<y,l_2>\ldots>,\ldots<z,l_3>\ldots\}$

  then $f_p(CONSTANTS(p)) = CONSTANTS(p) - <x,l_1> + <x,l_2\ op\ l_3>$

- If *p* has *n* statements then

  $f_p(CONSTANTS(p)) = f_n(f_{n-1}(f_{n-2}(\ldots f_2(f_1(CONSTANTS(p)))\ldots)))$

  where f_i is the function generated by the i^{th} statement in *p*

Constant Propagation over DEF-USE Chains

Worklist ← Ø

for i ← 1 to number of operations
  if in_1 of operation i is a constant c_i
    then Value(in_1,i) ← c_i
    else Value(in_1,i) ← T
  if in_2 of operation i is a constant c_j
    then Value(in_2,i) ← c_j
    else Value(in_2,i) ← T
  if (Value(in_1,i) is a constant &
    Value(in_2,i) is a constant)
    then Value(out,i) ← evaluate op i
      Worklist ← Worklist ∪ {i }
    else Value(out,i) ← T

Initialization Step

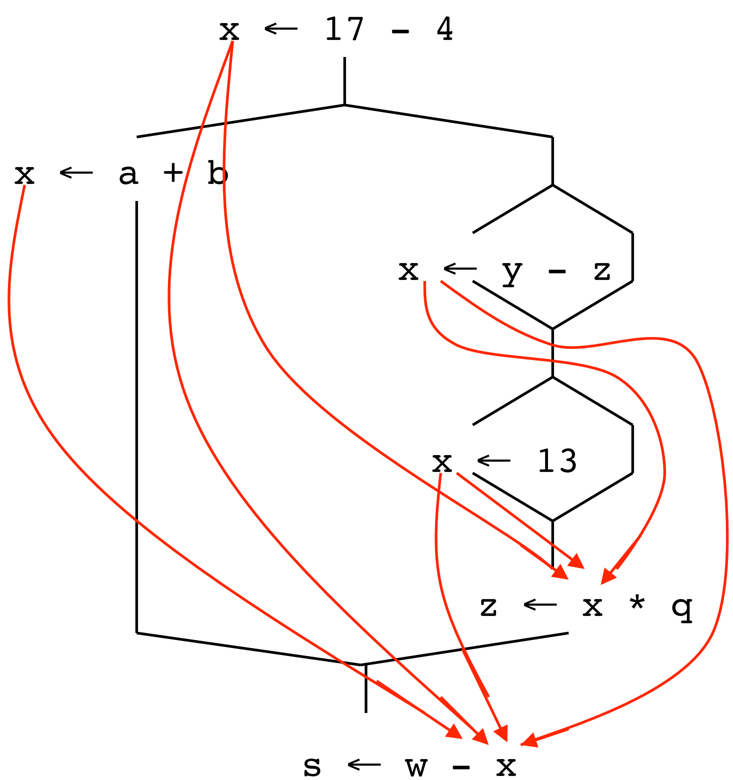
while (Worklist ≠ Ø)
  remove a definition i from WorkList
  for each j ∈ **USES**(out,i)
    let x be operand where j occurs
    Value(in_x,j) ← Value(in_x,j)
        ∧ Value(out,i)
    if (Value(in_1,j) is a constant &
      Value(in_2,j) is a constant)
      then Value(out,j) ← evaluate op j
        Worklist ← Worklist ∪ {j }
      else if (Value(in_1,j) is ⊥ or
          Value(in_2,j) is ⊥)
      then Value(out,j) ← ⊥
        Worklist ← Worklist ∪ {j }

Propagation Step

Any T left after fixed point derives from an uninitialized value. What should we do?

DEF-USE Chains

Example

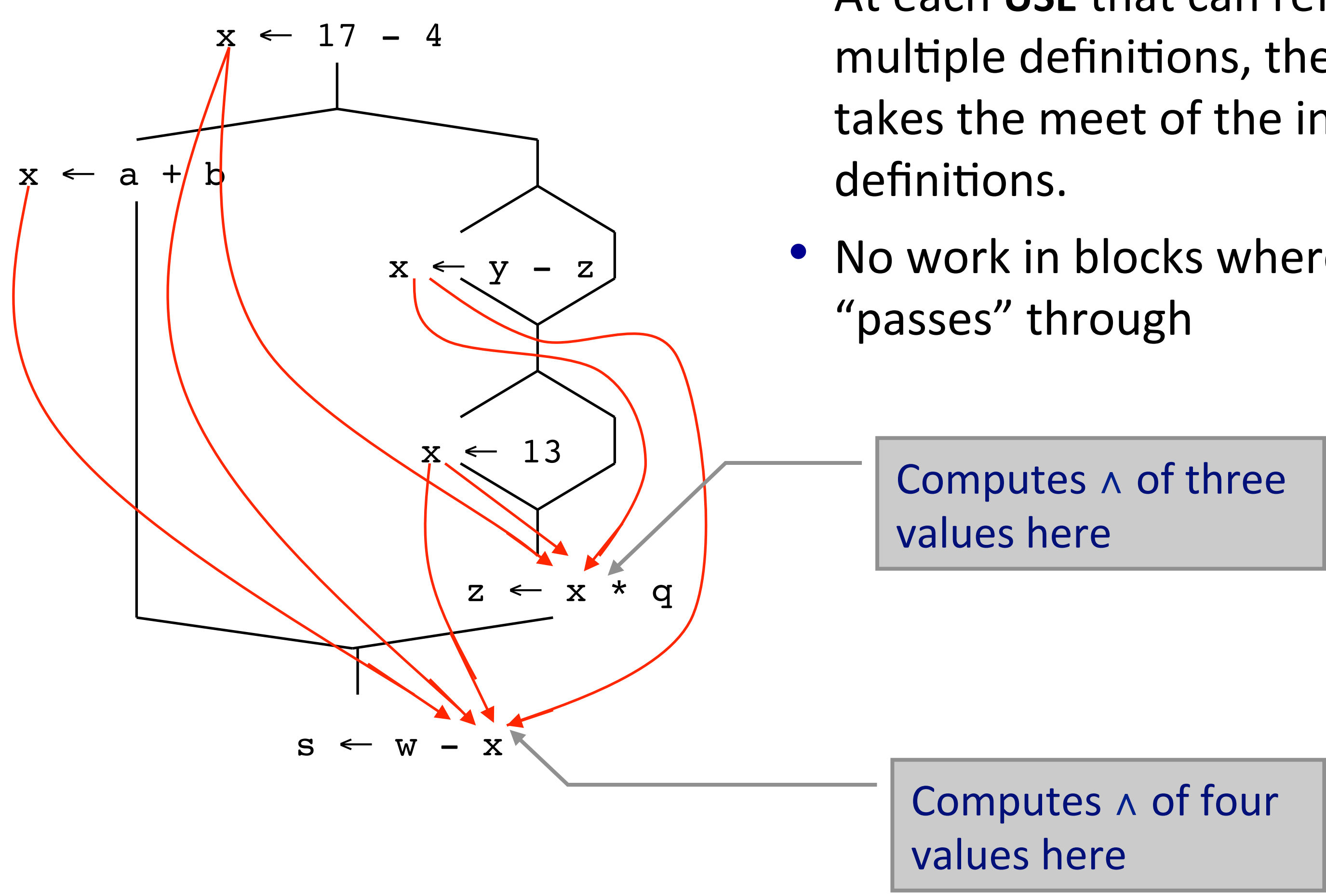


Applying the algorithm involves:

- Initialization step at each operation
  - → Two **DEF**s go on the worklist
  - → Others are not constant valued
- A multi-way meet at each use of *x*

Constant Propagation over DEF-USE Chains

Back to the Example



- At each **USE** that can refer to multiple definitions, the analysis takes the meet of the incoming definitions.
- No work in blocks where info just "passes" through

Constant Propagation over DEF-USE Chains

Complexity

- Initial step takes O(1) time per operation
- Propagation takes
  - ♦ |***USES***(*v,i*)| for each *i* pulled from Worklist
  - ♦ Summing over all ops, becomes |edges in **DEF-USE** graph|
  - ♦ A definition can be on the worklist twice (*lattice height*)
  - ♦ O(|operations| + |edges in **DEF-USE** graph|)

This sparse-graph[1] approach is faster than the straightforward iterative approach in the Kildall style — both in asymptotic complexity and in practical implementation.

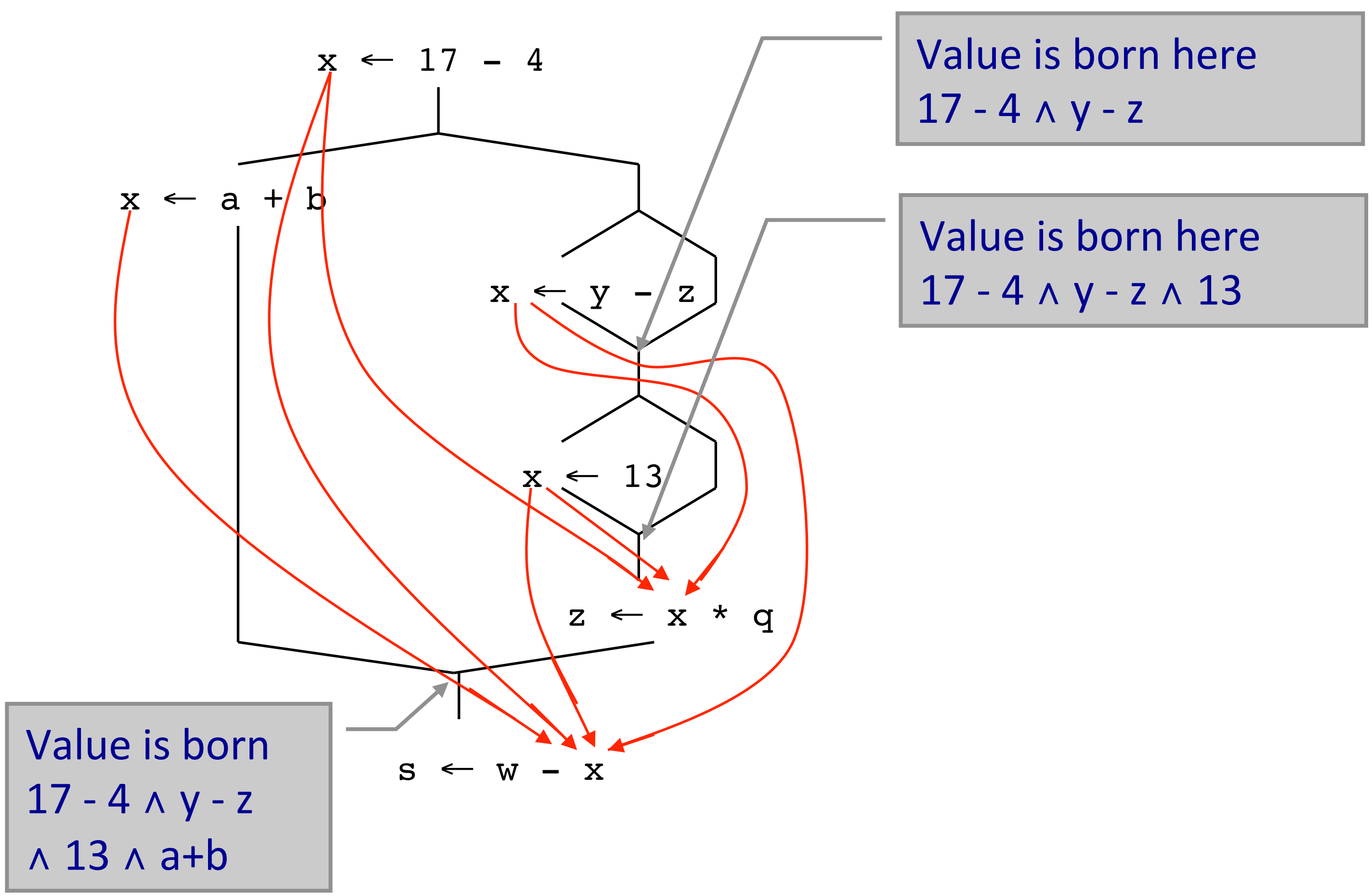
Still, the number of meets is **O**(|definitions|^2) in the worst case.

We can do better.

[1] We think of the **DEF-USE** graph as sparse because it connects the **DEF** directly to the **USE** without touching blocks in between them.

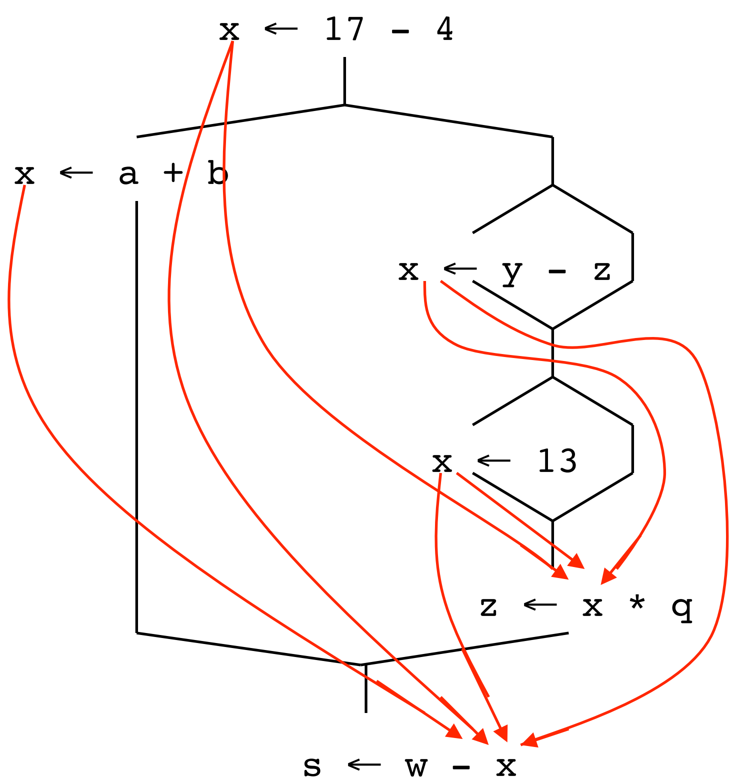
DEF-USE Chains and Birth Points

Birth Points Of Values



DEF-USE Chains and Birth Points

Birth Points Of Values



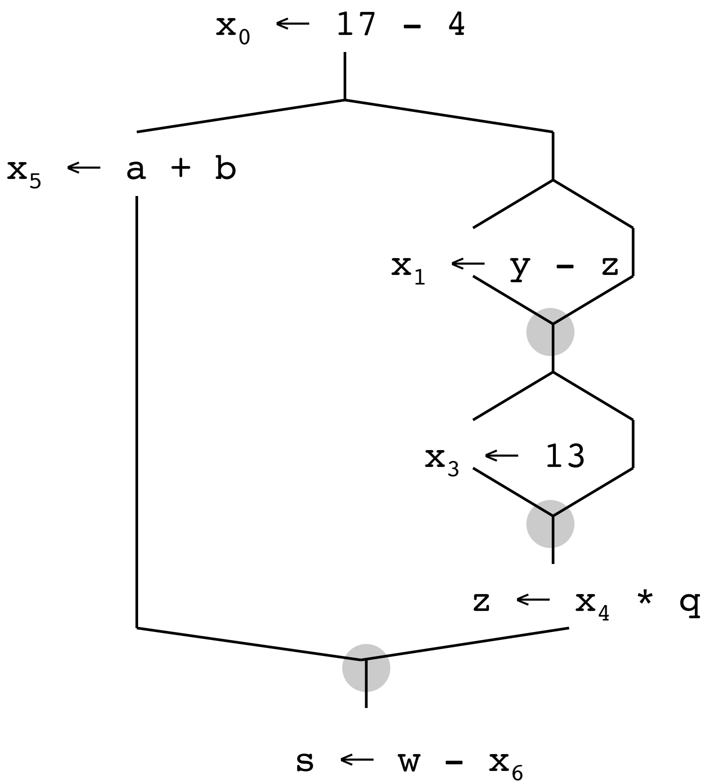
We should be able to compute the values that we need with fewer meet operations, if only we can find these birth points.

- Need to identify birth points
- Need to insert some artifact to force the evaluation to follow the birth points
- Enter Static Single Assignment form, or **SSA**

Essentially, we want a **DEF-USE** graph that has fewer edges.

DEF-USE Chains and Birth Points

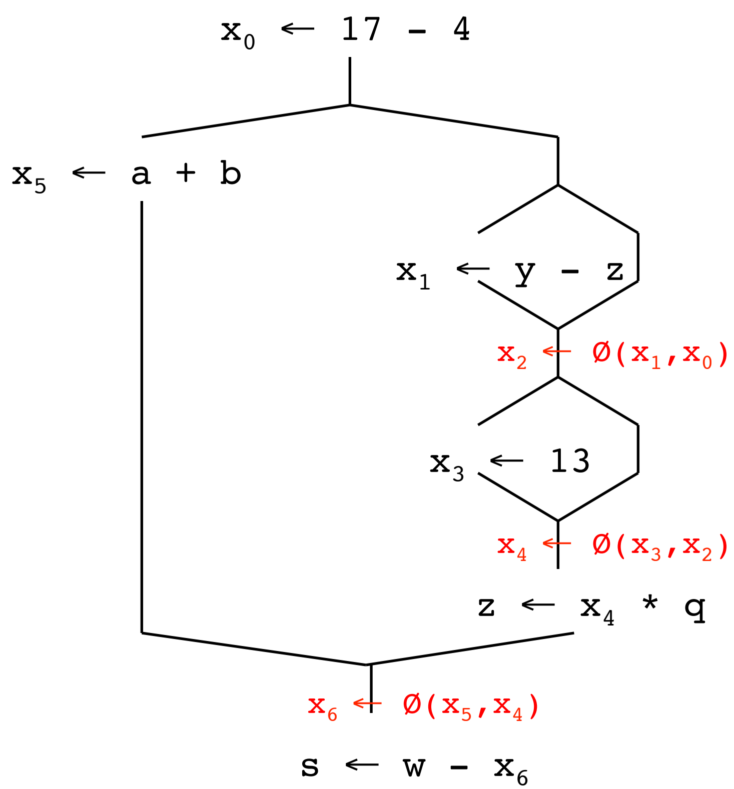
Making Birth Points Explicit



There are three birth points for x

DEF-USE Chains and Birth Points

Making Birth Points Explicit



Each birth point needs a definition to reconcile the values of x

- Insert a ø-function at each birth point
- Rename values so each name is defined once
- Now, each use refers to one definition

⇒ Static Single-Assignment Form

Building Static Single-Assignment Form

SSA Form

- Each name is defined exactly once
- Each use refers to exactly one name

What's hard

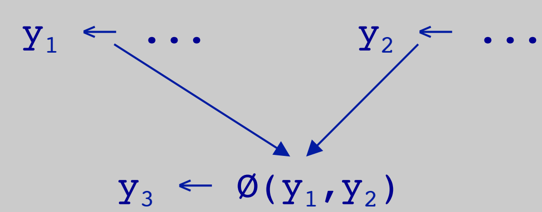
- Straight-line code is trivial
- Splits in the CFG are trivial
- Joins in the CFG are hard

Building SSA Form

- Insert ϕ-functions at birth points of values
- Rename all values for uniqueness

A ϕ-function is a special kind of copy that selects one of its parameters.

The choice of parameter is governed by the CFG edge along which control reached the current block.



I know of no machine that implements a ϕ-function directly in hardware.

SSA Construction Algorithm (High-level sketch)

1. Insert ϕ-functions
2. Rename values

… that's all …

… of course, there is some bookkeeping to be done …

SSA Construction Algorithm (The naïve algorithm)

1. Insert ϕ-functions at every join[1] for every name
2. Solve *reaching definitions*
3. Rename each use to the def that reaches it (*will be unique*)

Builds a version of SSA with the maximal number of ϕ- functions

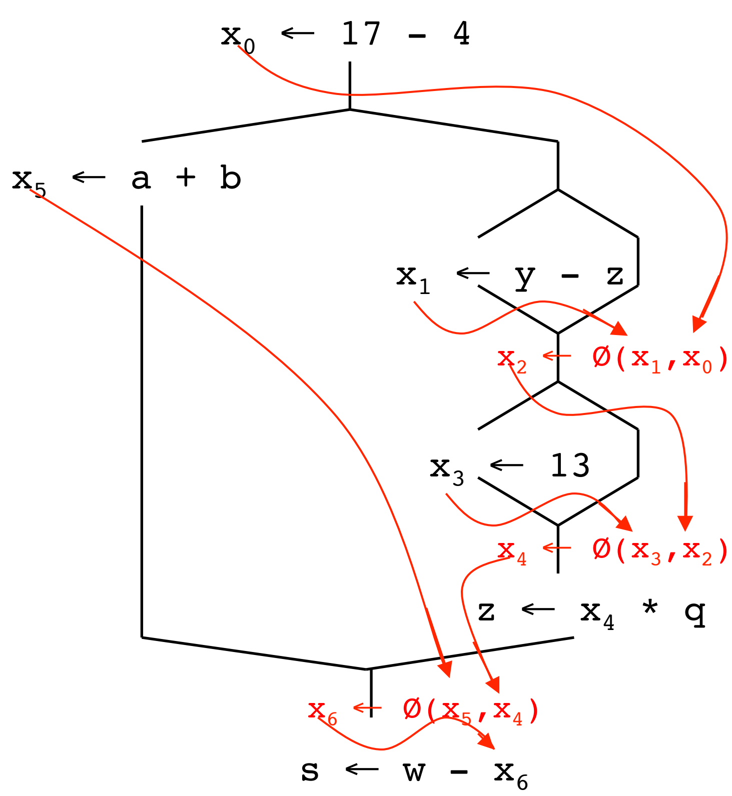
What's wrong with this approach

- Too many ϕ-functions (*precision*)
- Too many ϕ-functions (*space*)
- Too many ϕ-functions (*time*)
- Need to relate edges to ϕ-functions parameters (*bookkeeping*)

To do better, we need a more complex approach

[1] Every birth point occurs at a definition or a join point in the **CFG**.

Back to the Example and Birth Points



The naïve algorithm inserts too many ø functions

- Our goal was a ø-function at each birth point
- Naïve algorithm inserts a ø for each name at each merge in the CFG

The naïve algorithm produces

- Correct SSA form
- More ø's than any other known algorithm for SSA construction

The rest is optimization (!)

Key Point: number of meet operations that constant propagation performs is now a property of both placement of definitions & CFG structure. In practice, we expect to perform many fewer meets & to see that the number of meets grows more slowly.

SSA Construction Algorithm (Detailed sketch for pruned SSA)

1. Insert ϕ-functions
   a. calculate dominance frontiers
   b. find global names

      for each name, build a list of blocks that define it

   c. insert ϕ-functions

      ∀ global name *n*

      ∀ block *b* in which *n* is assigned

      ∀ block *d* in *b*'s dominance frontier

      insert a ϕ-function for *n* in *d*

      add *d* to *n*'s list of defining blocks

Critical, but moderately complex; DFs guide ϕ-function insertion

Compute list of blocks where each name is assigned & use as a worklist

This adds to the worklist !

Creates the iterated dominance frontier

Use a checklist to avoid putting blocks on the worklist twice; keep another checklist to avoid inserting the same ϕ-function twice.

SSA Construction Algorithm (Detailed sketch)

2. Rename variables in a pre-order walk over dominator tree

   (use an array of stacks, one stack per global name)

   Staring with the root block, *b*

   1 counter per name for subscripts

   a. generate unique names for result of each ϕ-function and push them on the appropriate stacks
   b. rewrite each operation in the block
      i. Rewrite uses of global names with the current version (from the stack)
      ii. Rewrite definition by inventing & pushing new name
   c. fill in ϕ-function parameters of successor blocks
   d. recurse on *b*'s children in the dominator tree
   e. <on exit from block *b*> pop names generated in *b* from stacks

Reset the state

Need the end-of-block name for this path

Dominance Frontiers & Inserting ϕ-functions

Where does an assignment in block *n* induce a ϕ–function?

- *n* DOM *m* $\Rightarrow$ no need for a ϕ–function in *m*
  - ♦ Definition in *n* blocks any previous definition from reaching *m*
- If *m* has multiple predecessors, and *n* dominates one of them, but not all of them, then *m* needs a ϕ–function for each definition in *n*

More formally, *m* is in the dominance frontier of *n* if and only if

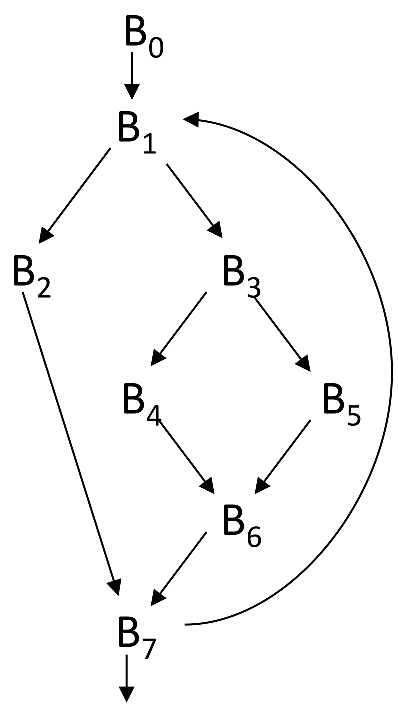
1. $\exists\, p \in preds(m)$ such that $n \in$ DOM(p), and
2. *n* does not *strictly dominate* *m* ($n \notin$ DOM$(m) - \{ m \}$)

This dominance frontier is precisely what we need to insert ϕ–functions:

A def in block n induces a ϕ–function in each block in DF(n).

"Strict" dominance allows a ϕ–function at the head of a single-block loop.

DOM Example

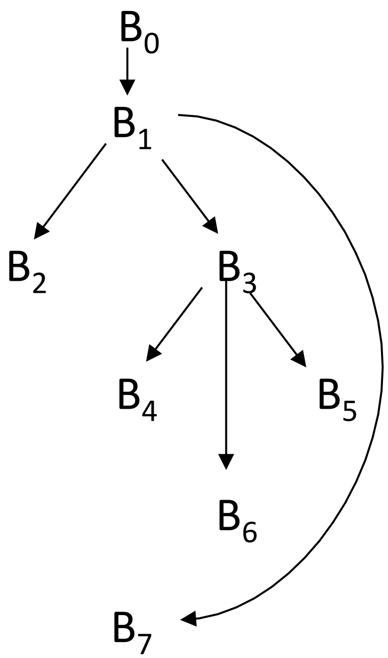


Flow Graph

Results of iterative solution for DOM

	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
IDOM	0	0	1	1	3	3	3	1

Example

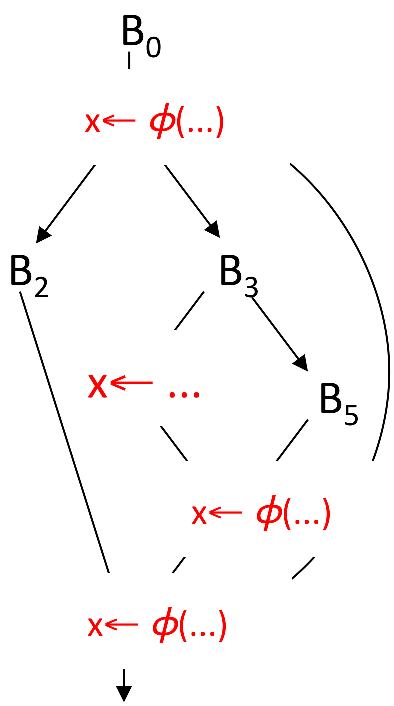


Dominance Tree

Results of iterative solution for DOM

	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
IDOM	0	0	1	1	3	3	3	1

Example



Dominance Frontiers

Dominance Frontiers & ϕ-Function Insertion

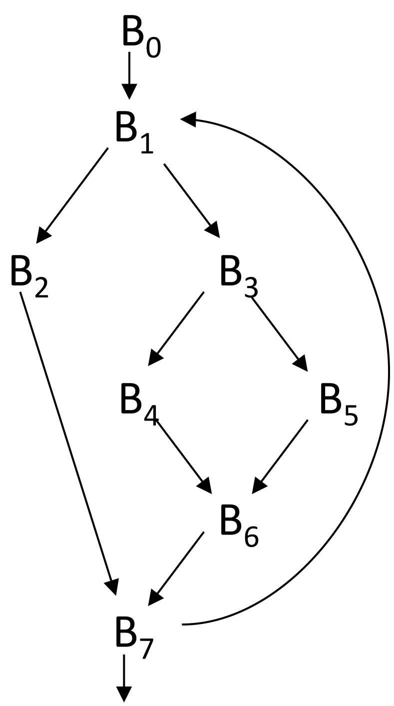
- A definition at *n* forces a ϕ-function at *m* iff $n \notin \text{Dom}(m)$ but $n \in \text{Dom}(p)$ for some $p \in preds(m)$
- DF(*n*) is fringe just beyond region *n* dominates

	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
Strict DF	1	1	7	7	6	6	7	1

- DF(4) is {6}, so ← in 4 forces ϕ-function in 6
- ← in 6 forces ϕ-function in DF(6) = {7}
- ← in 7 forces ϕ-function in DF(7) = {1}
- ← in 1 forces ϕ-function in DF(1) = {1}
  (halt – the ϕ is already there)

For each assignment, we insert the ϕ-functions

Example



Dominance Frontiers

	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
Strict DF	1	1	7	7	6	6	7	1

Computing Dominance Frontiers

- 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* (*i.e., |preds(x)|* > 1)

  For each **CFG** predecessor *p* of *x*

  Run from p to **IDOM**(*x*) *in the dominator tree*, & add *x* to **DF**(*n*) for each *n* from *p* up to but not **IDOM**(*x*)

- For some applications (other than building **SSA**), we need post-dominance, the post-dominator tree, and reverse dominance frontiers, **RDF**(*n*)
  - Just dominance on the reverse **CFG**
  - Reverse the edges & add unique exit node
- We will use these ideas in dead code elimination

SSA Construction Algorithm (Reminder)

1. Insert ϕ-functions at every join for every name
   a. calculate dominance frontiers
   b. find global names

      for each name, build a list of blocks that define it
   c. insert ϕ-functions

A “global” is LIVE on input to some block
x is global iff $\exists b \ni x \in$ UEVAR(b)

$\forall$ global name n
  $\forall$ block b in which n is assigned
    $\forall$ block d in b’s dominance frontier
      insert a ϕ-function for n in d
      add d to n’s list of defining blocks

Step 1.b is not in the CFRWZ [110] algorithms
It produces an SSA form with fewer ϕ-functions [50]

SSA Construction Algorithm

Finding global names

- Difference between different forms of **SSA**
- Minimal SSA uses all names **[CFRWZ, 110]**
- Semi-pruned uses names that are *live* on entry to some block **[50]**
  - ♦ Shrinks name space & number of ϕ-functions
  - ♦ Pays for itself in compile-time speed
- For each "global name", need a list of blocks where it is defined
  - ♦ Drives ϕ-function insertion
  - ♦ b defines x implies a ϕ-function for x in every $c \in \mathrm{DF}(b)$

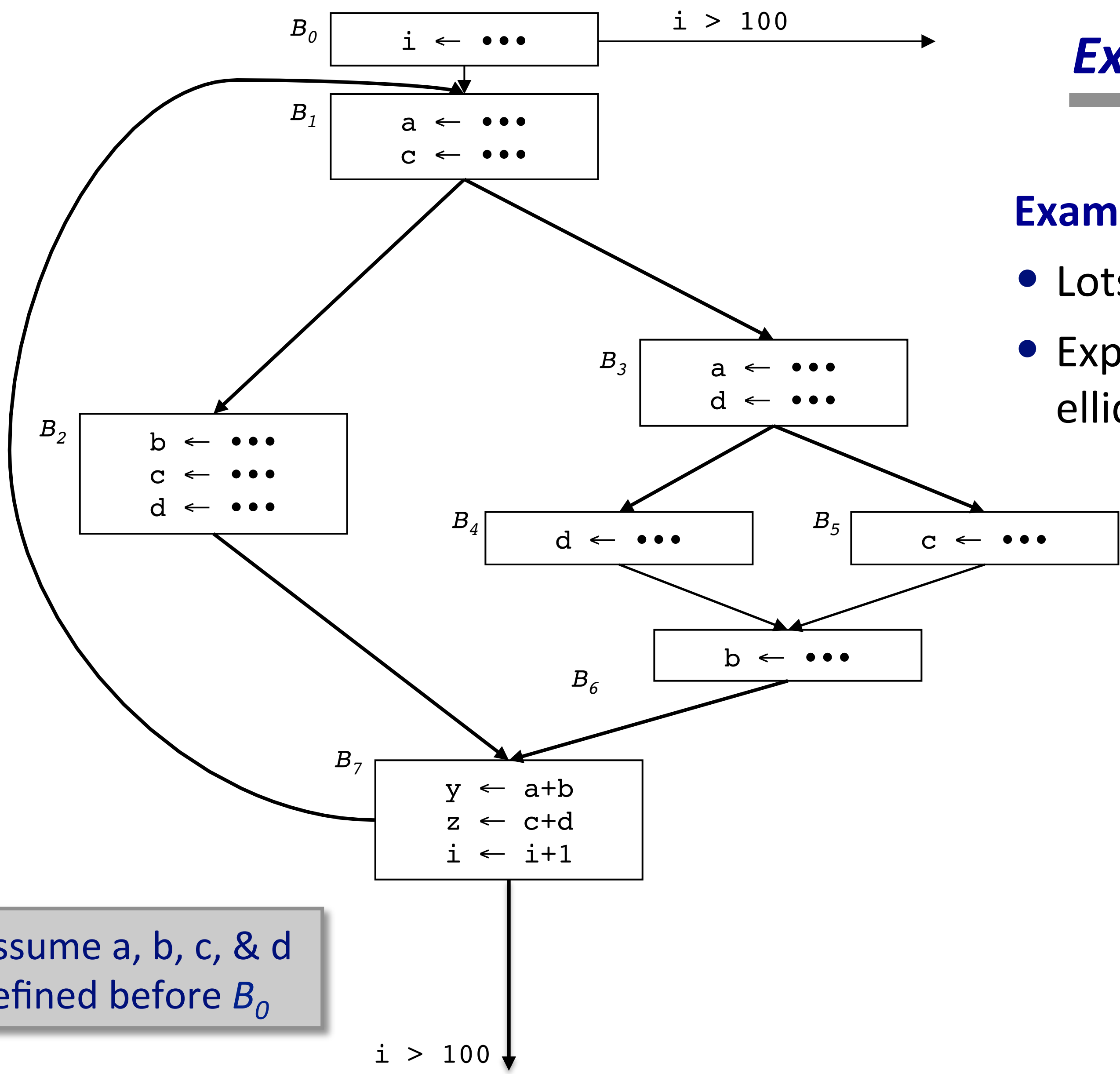
Otherwise, needs no ϕ-function.
Can use local notion of *live.*

Pruned SSA adds a test to see if x is live at insertion point

Occasionally, building pruned is faster than building semi-pruned.

Any algorithm that has non-linear behavior in the number of ϕ-functions will have a size where pruned is the SSA flavor of choice.

Example

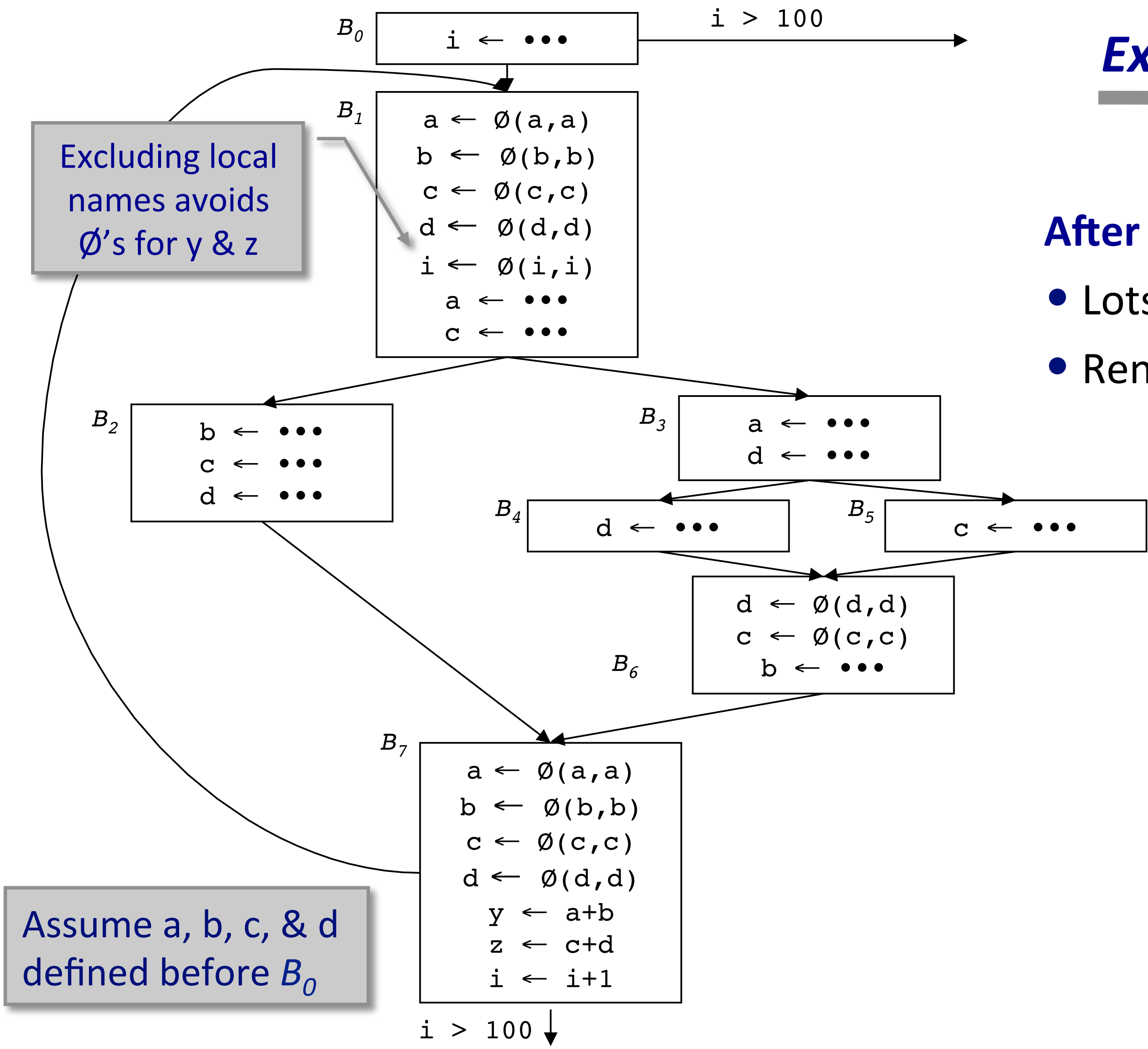


Example CFG

- Lots of assignments
- Expression details ellided

Assume a, b, c, & d defined before B_0

Example



After Ø insertion

- Lots of new ops
- Renaming is next

SSA Construction Algorithm

One Final Point About *Φ*-function Insertion

- *Φ*-functions have an unusual semantics
  - ♦ When execution enters a block, all *Φ*-functions evaluate their arguments, *in parallel*, and then perform their assignments, *in parallel*
  - ♦ This behavior allows the compiler to manipulate *Φ*-functions without worrying about the order in which they appear at the head of a block
- The parallel semantics of *Φ*-functions will introduce complications when the compiler tries to translate code in SSA form back into executable code

SSA Construction Algorithm (Detailed sketch)

2. Rename variables in a pre-order walk over dominator tree

   (use an array of stacks, one stack per global name)

   Starting with the root block, *b*

   a. generate unique names for each ϕ-function and push them on the appropriate stacks

   b. rewrite each operation in the block

      i. Rewrite uses of global names with the current version (from the stack)

      ii. Rewrite definition by inventing & pushing new name

   c. fill in ϕ-function parameters of successor blocks

   d. recurse on *b*'s children in the dominator tree

   e. <on exit from block *b*> pop names generated in *b* from stacks

1 counter per name for subscripts

Reset the state

Need the end-of-block name for this path

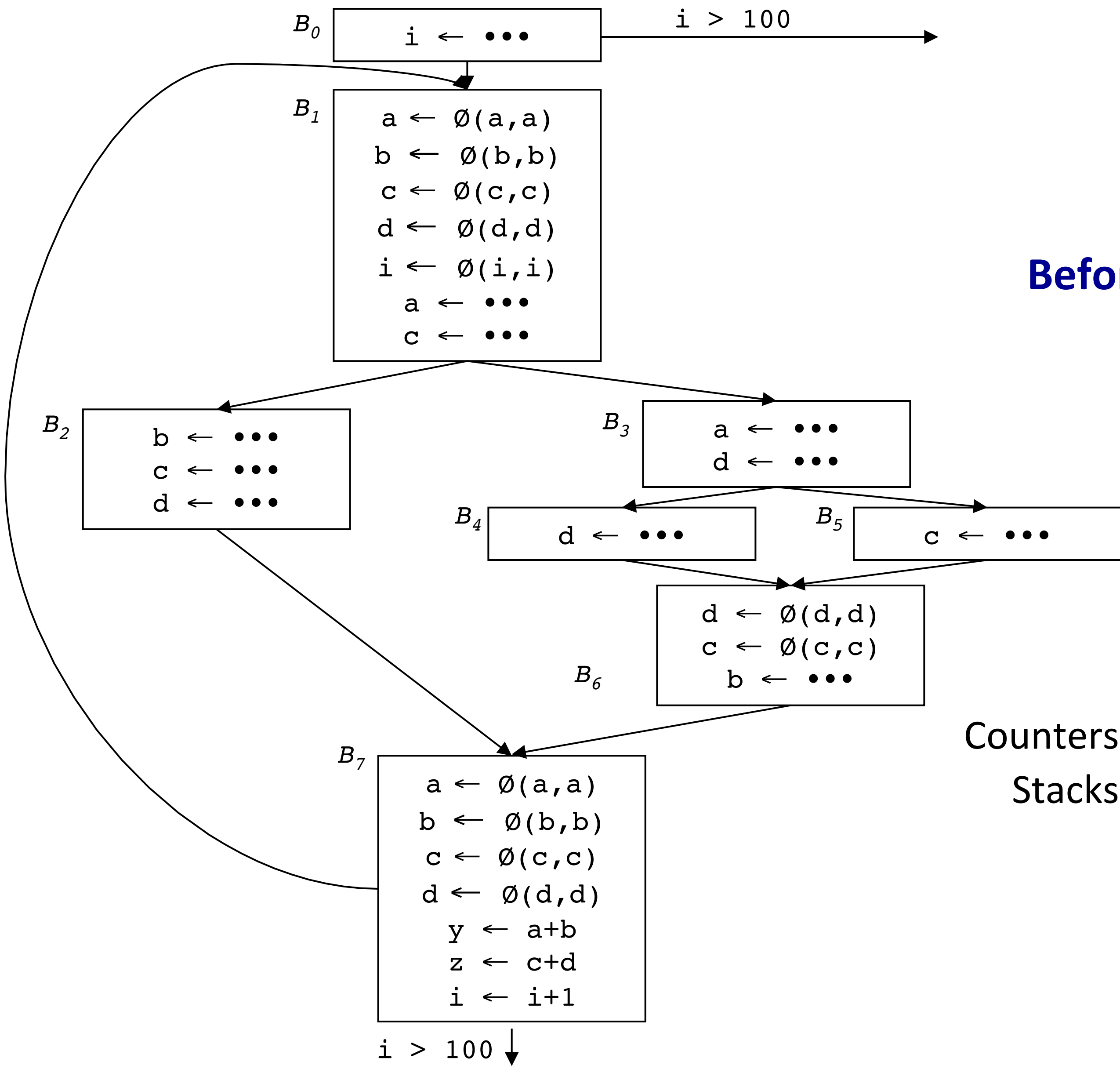
SSA Construction Algorithm (Less high-level sketch)

Adding the details ...

```
for each global name i
   counter[i] ← 0
   stack[i] ← Ø
call Rename(n0)

NewName(n)
   i ← counter[n]
   counter[n] ← counter[n] + 1
   push ni onto stack[n]
   return ni
```

```
Rename(b)
   for each φ-function in b, x ← φ (...)
      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 φ parameters

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

   for each operation "x ← y op z" in b
            or each phi-function
      pop(stack[x])
```

Minor engineering nit: assume, up front, that we convert all names into unique small integers



Example

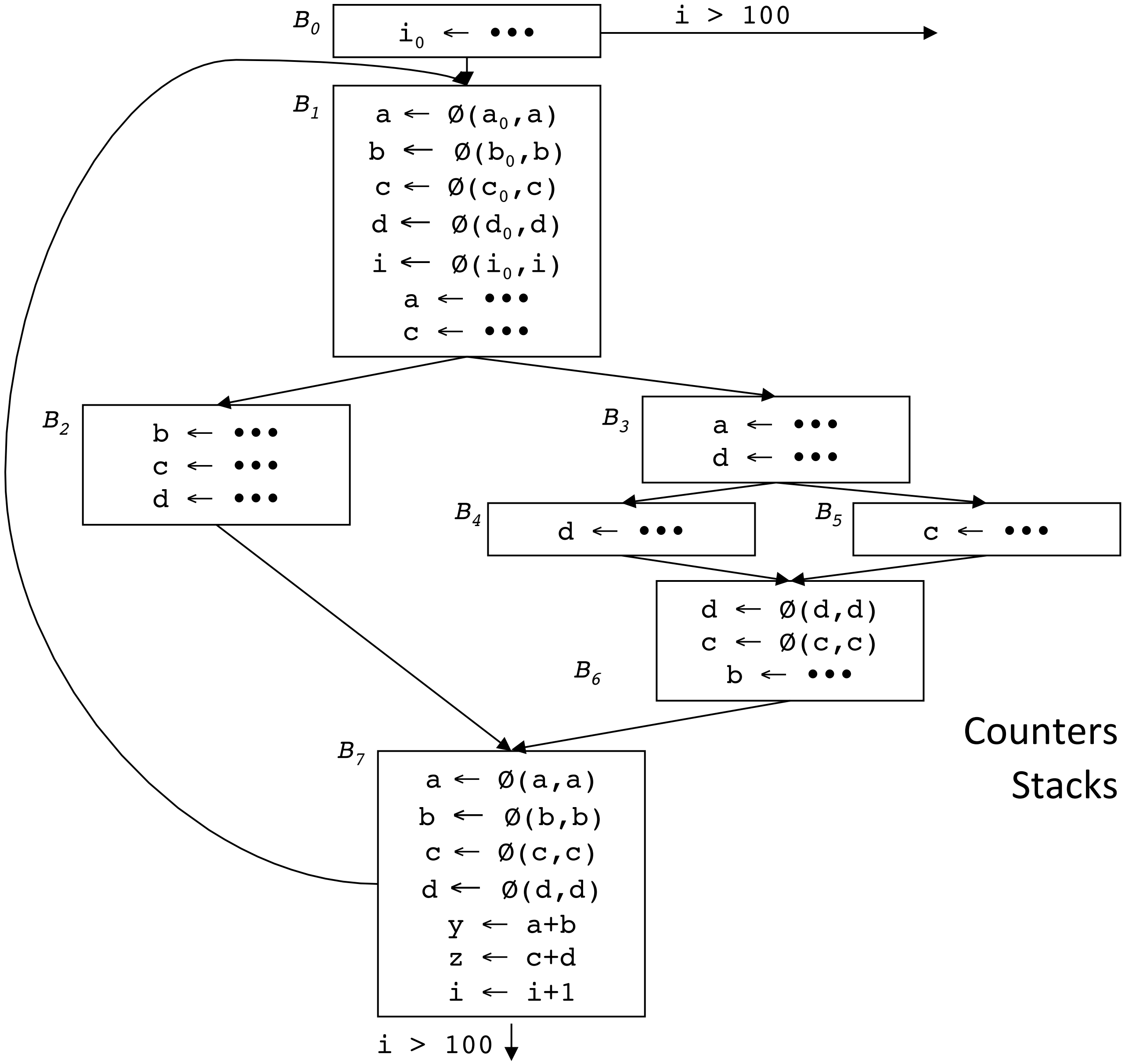
Before processing B_0

Assume a, b, c, & d defined before B_0

	a	*b*	*c*	*d*	*i*
Counters	1	1	1	1	0
Stacks	a_0	b_0	c_0	d_0	

i has not been defined

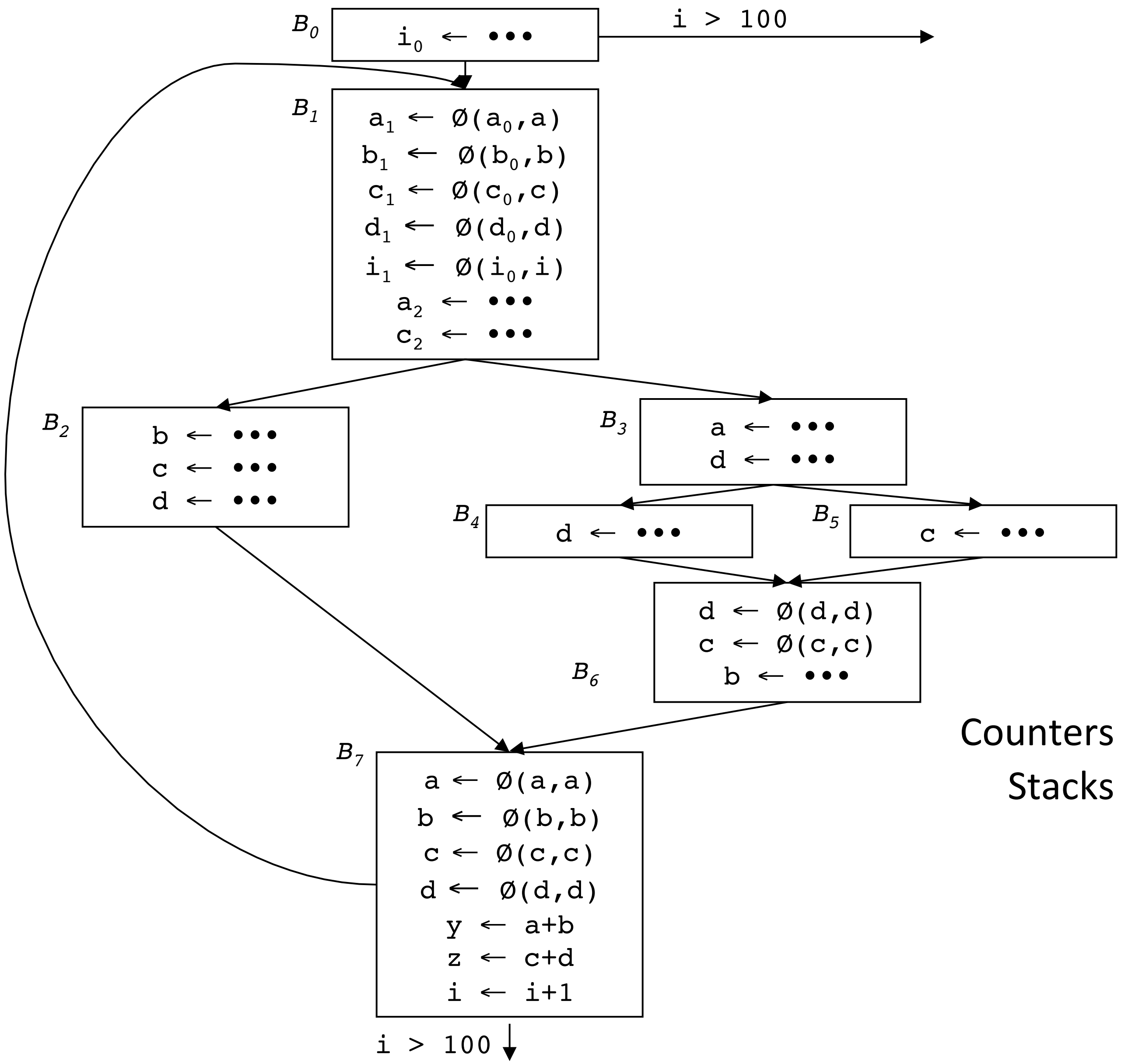
Example



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

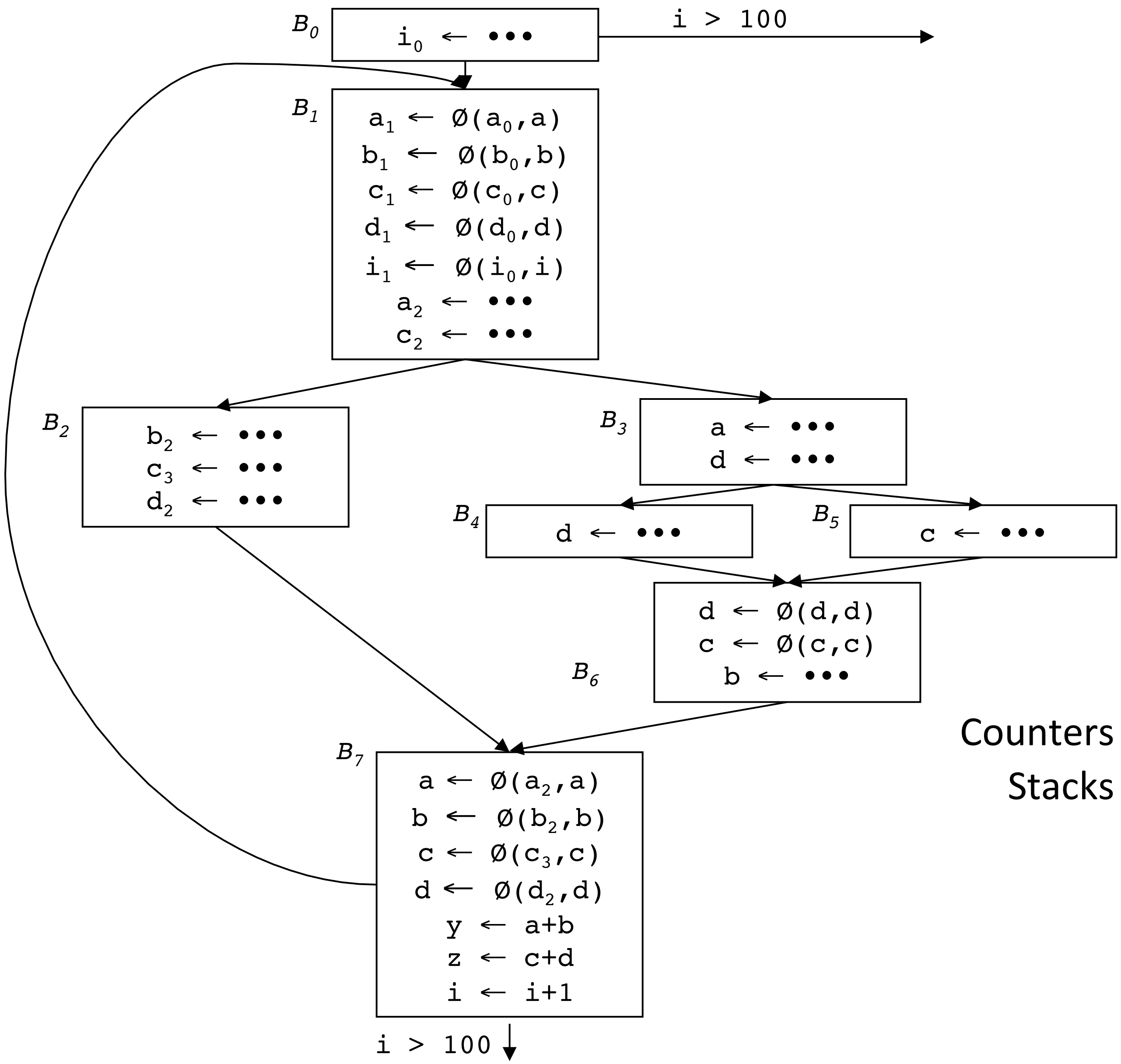


End of B_1

	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		

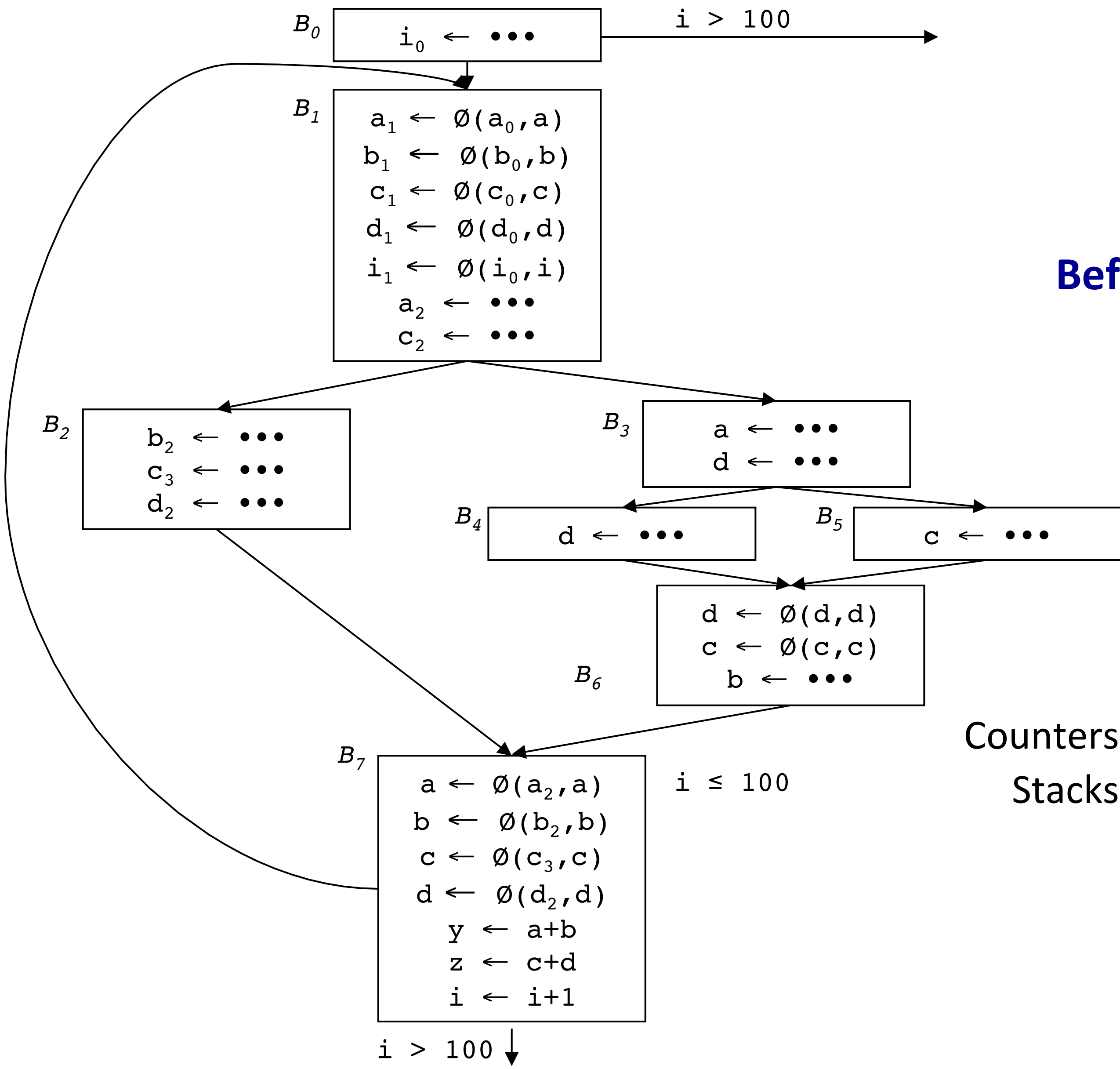
Example

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

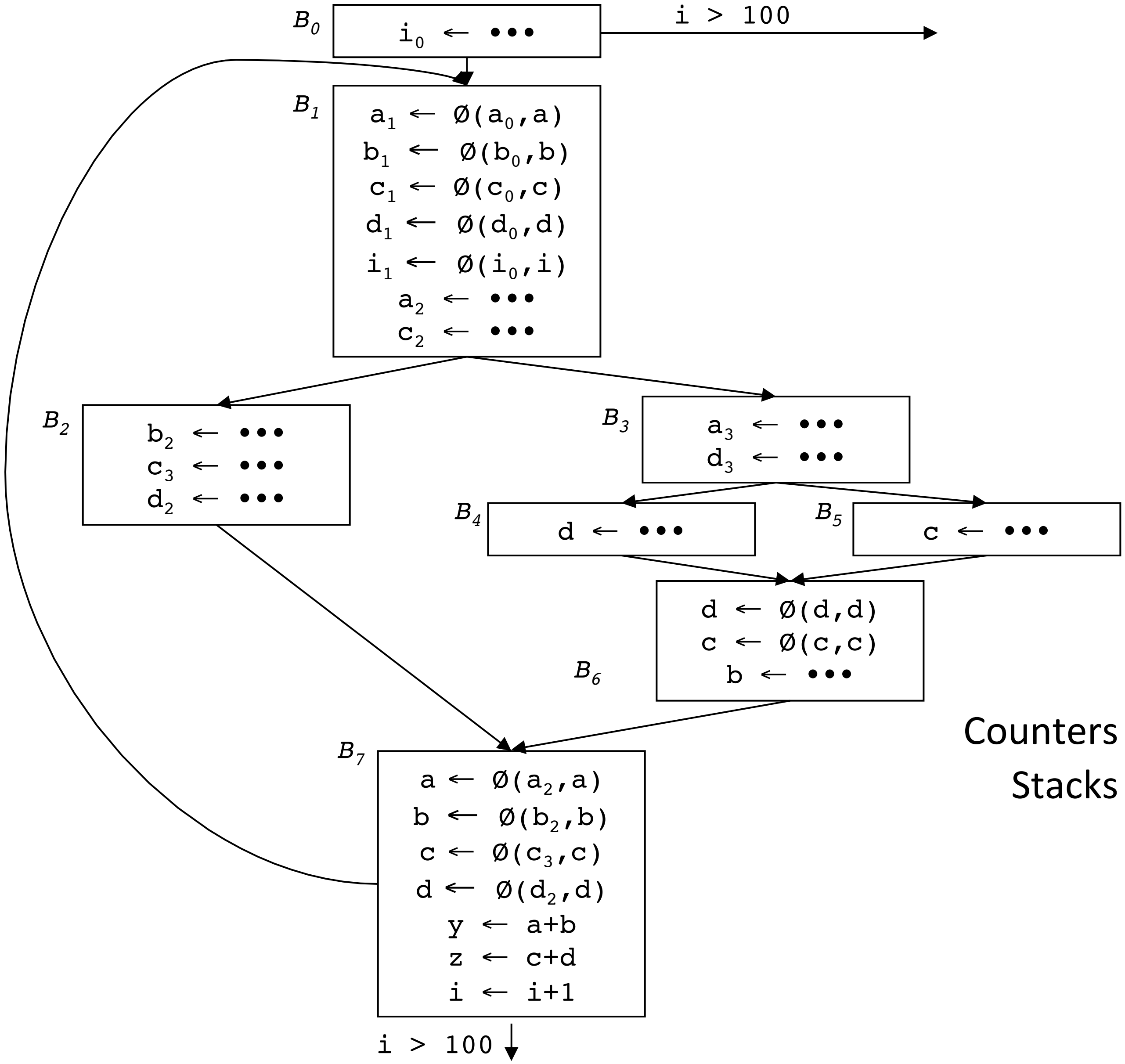


Before starting B_3

	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

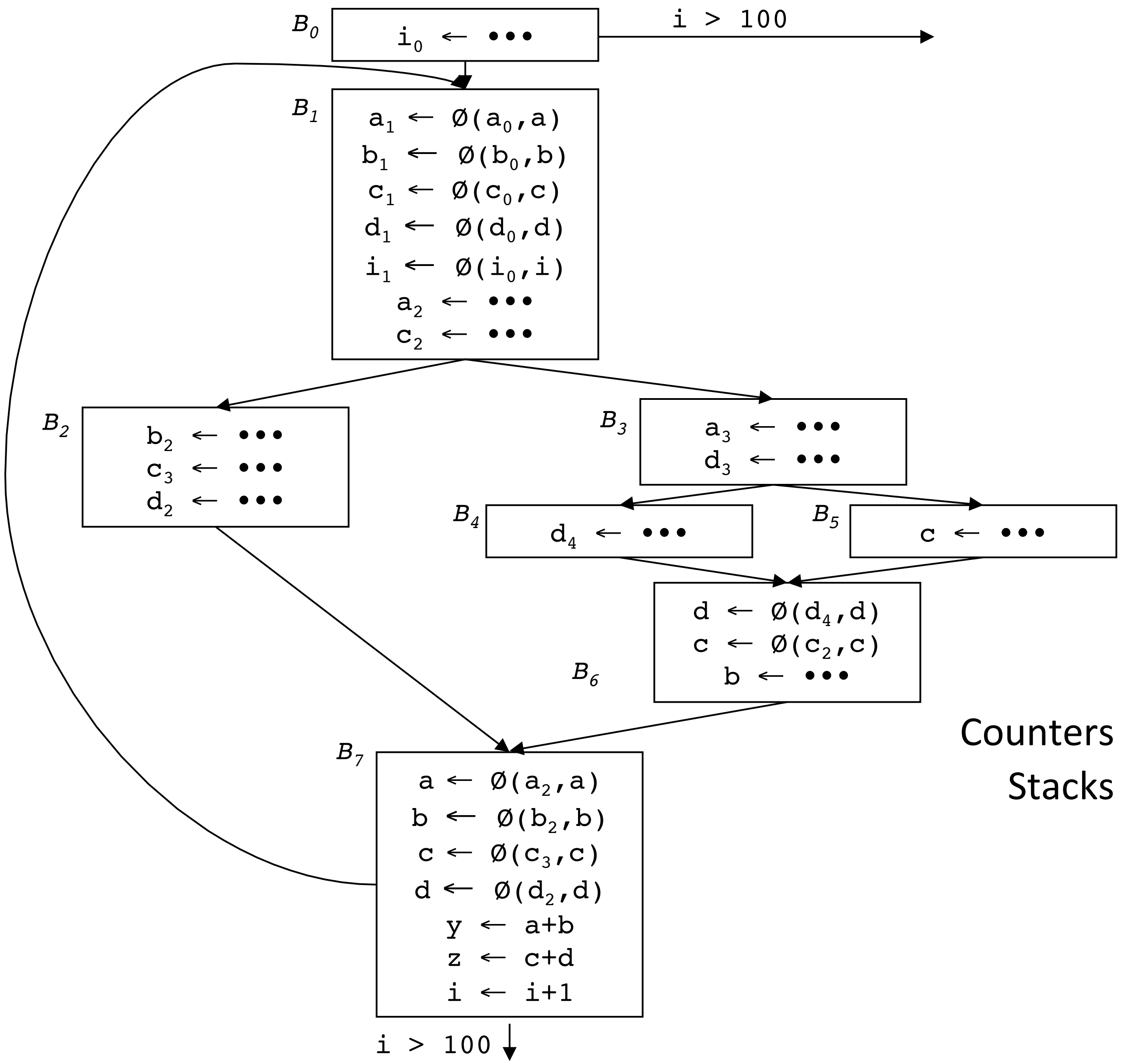
End of B_3



	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

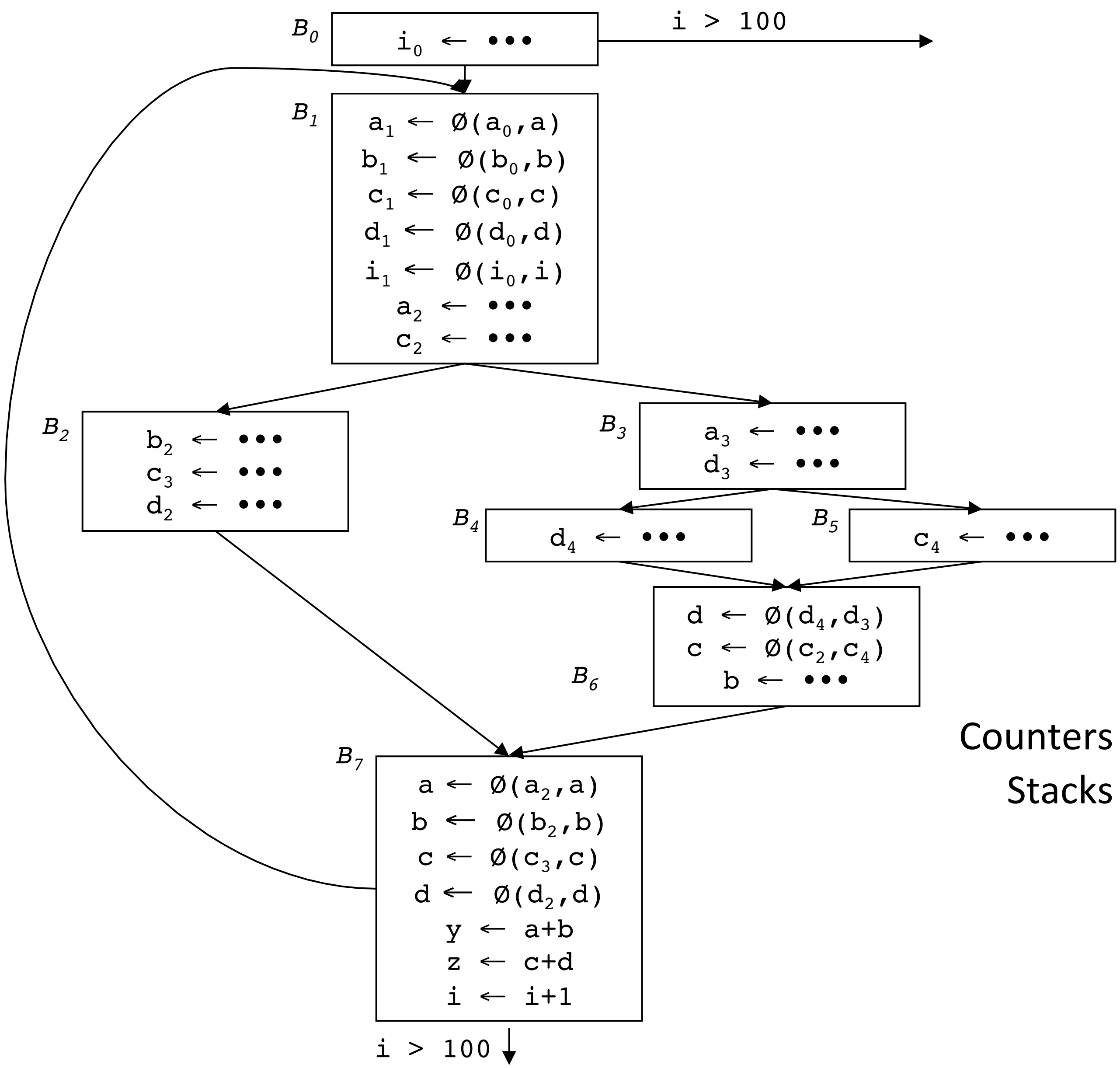
End of *B₄*



	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	

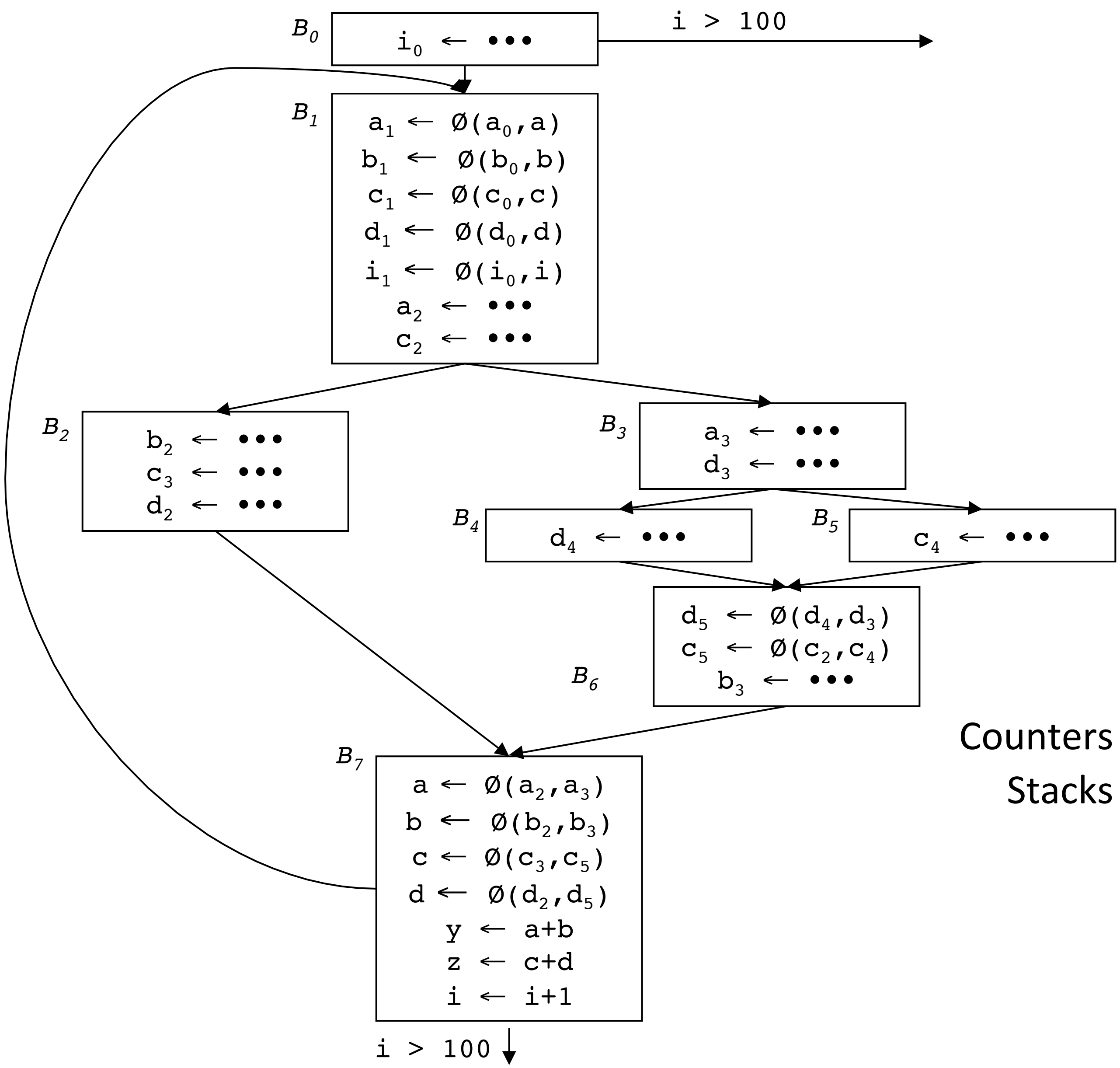
Example

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

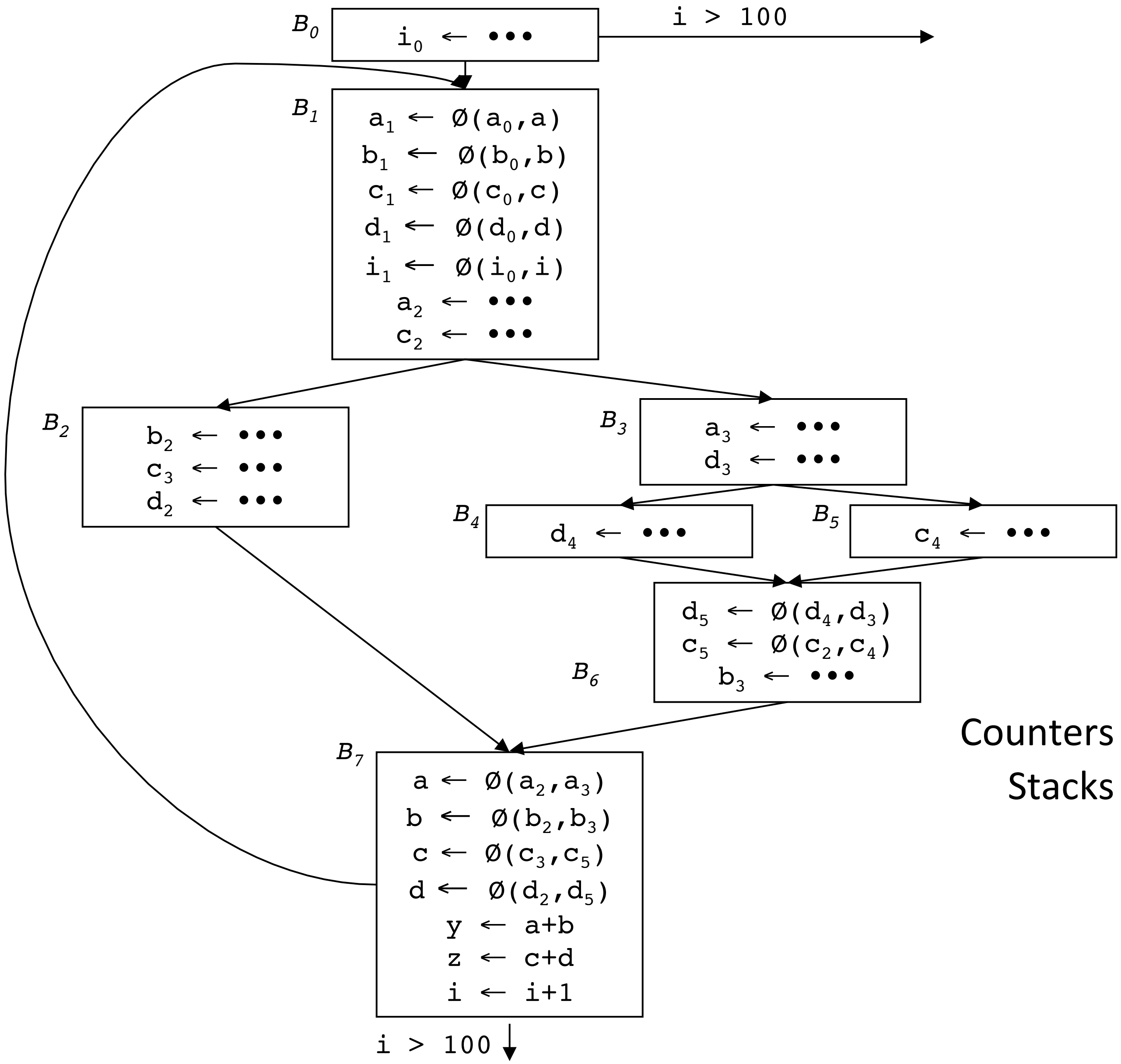


End of B_6

	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	

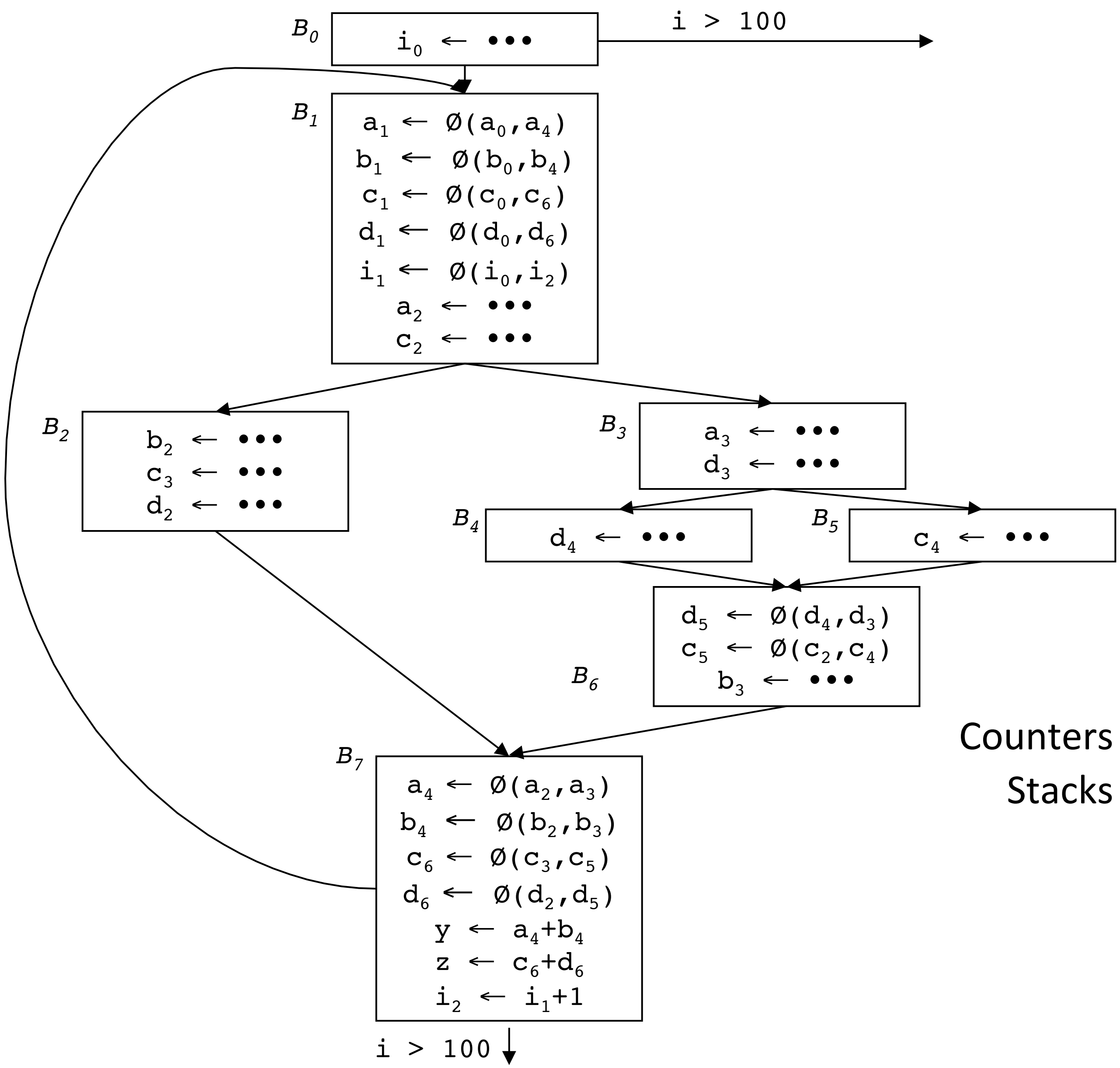
Example

Before *B₇*



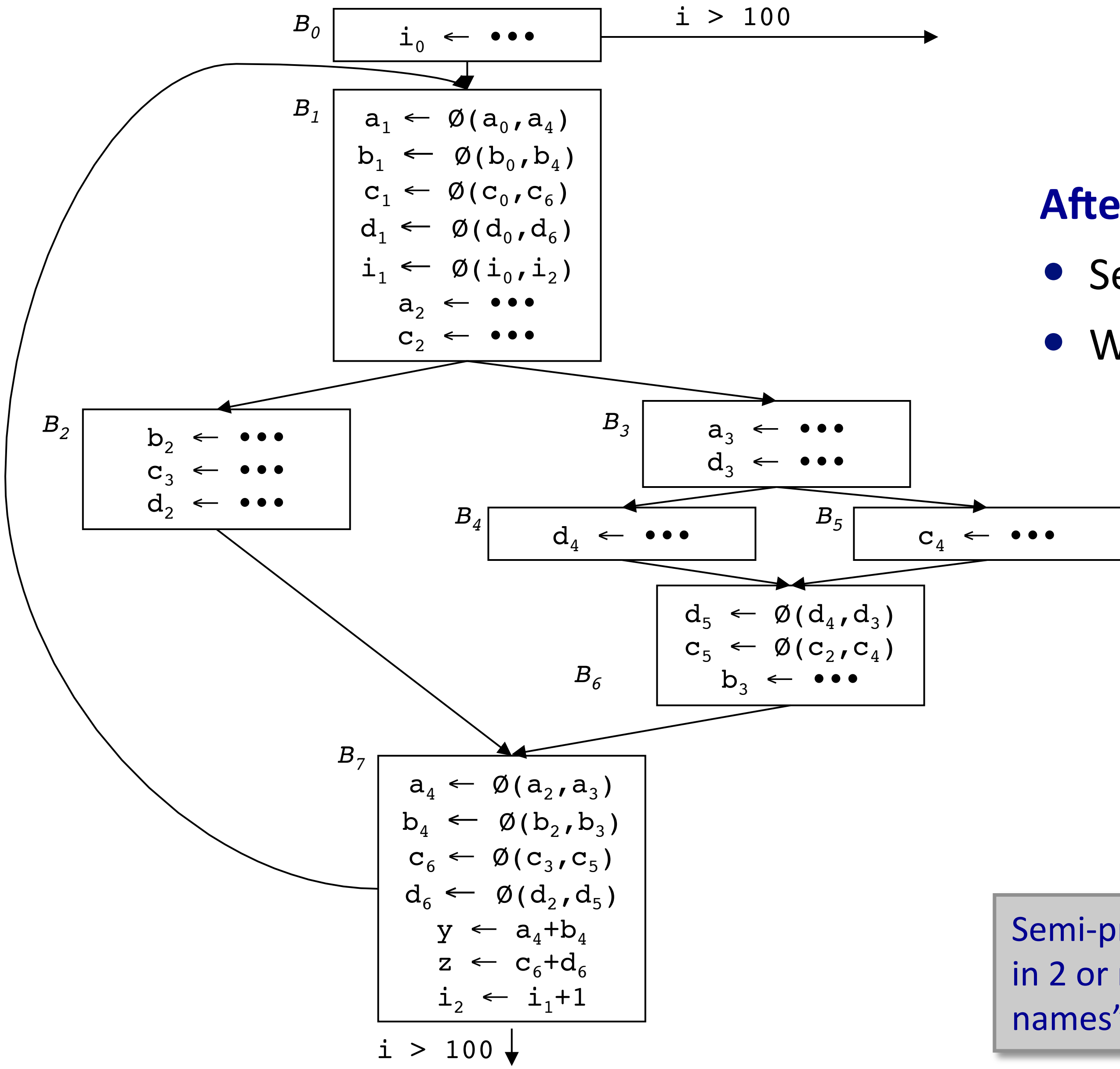
	a	*b*	*c*	*d*	*i*
Counters	4	4	6	6	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

Example



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		



Example

After renaming

- Semi-pruned SSA form
- We're done …

Semi-pruned ⇒ only names live in 2 or more blocks are "global names".

SSA Construction Algorithm (Pruned SSA)

What's this "pruned SSA" stuff?

- Minimal SSA still contains extraneous ϕ-functions
- Inserts some ϕ-functions where they are dead
- Would like to avoid inserting them

Two ideas

- *Semi-pruned SSA*: discard names used in only one block [50]
  - Significant reduction in total number of ϕ-functions
  - Needs only local Live information (*cheap to compute*)
- *Pruned SSA*: only insert ϕ-functions where their value is live [1]
  - Inserts even fewer ϕ-functions, but costs more to do
  - Requires computation of *LIVE* sets (*more expensive*)

In practice, both are simple modifications to step 1.

[1] J.D. Choi, R. Cytron, & J. Ferrante, "Automatic construction of sparse data flow evaluation graphs," POPL 91, pages 55-66.

SSA Construction Algorithm

We can improve the stack management

- Push at most one name per stack per block (*save push & pop*)
- Thread names together by block
- To pop names for block *b*, use *b*'s thread

This is another good use for a scoped hash table

- Significant reductions in pops and pushes
- Makes a minor difference in SSA construction time
- Scoped table is a clean, clear way to handle the problem