CS 4240: Compilers

Lecture 6: Static Single Assignment (SSA) Form

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

January 28, 2019

REMINDERS

- Homework 1 was released on Monday (1/14/19) on Piazza
  - Due by 11:59pm on Wednesday, 1/30/19 on Canvas
  - Must be submitted as PDF file
  - 5% of course grade

- Project 1 was released on Wednesday (1/16/19) on Piazza
  - Due by 11:59pm on Wednesday, 2/13/19 on Canvas
  - Must be submitted as zip file including instructions on how to build and run your project
  - 5% of course grade
  - Project teams have been announced on Piazza — please inform us ASAP of any inaccuracies

- MIDTERM EXAM: Wednesday, March 13, 4:30pm - 5:45pm

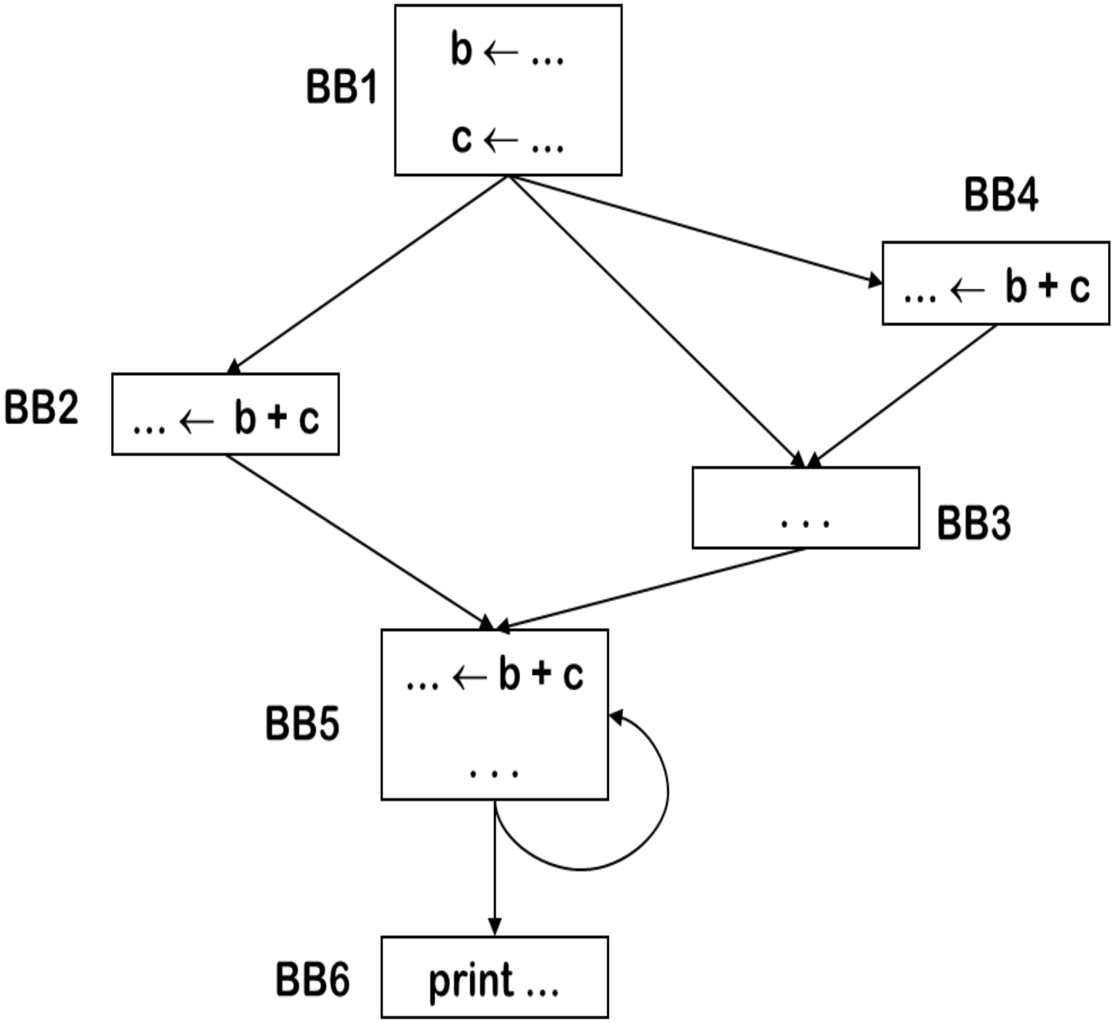
- FINAL EXAM: Wednesday, May 1 2:40 PM _ 5:30 PM

Worksheet–5 Solution

From lecture given on 01/23/2019

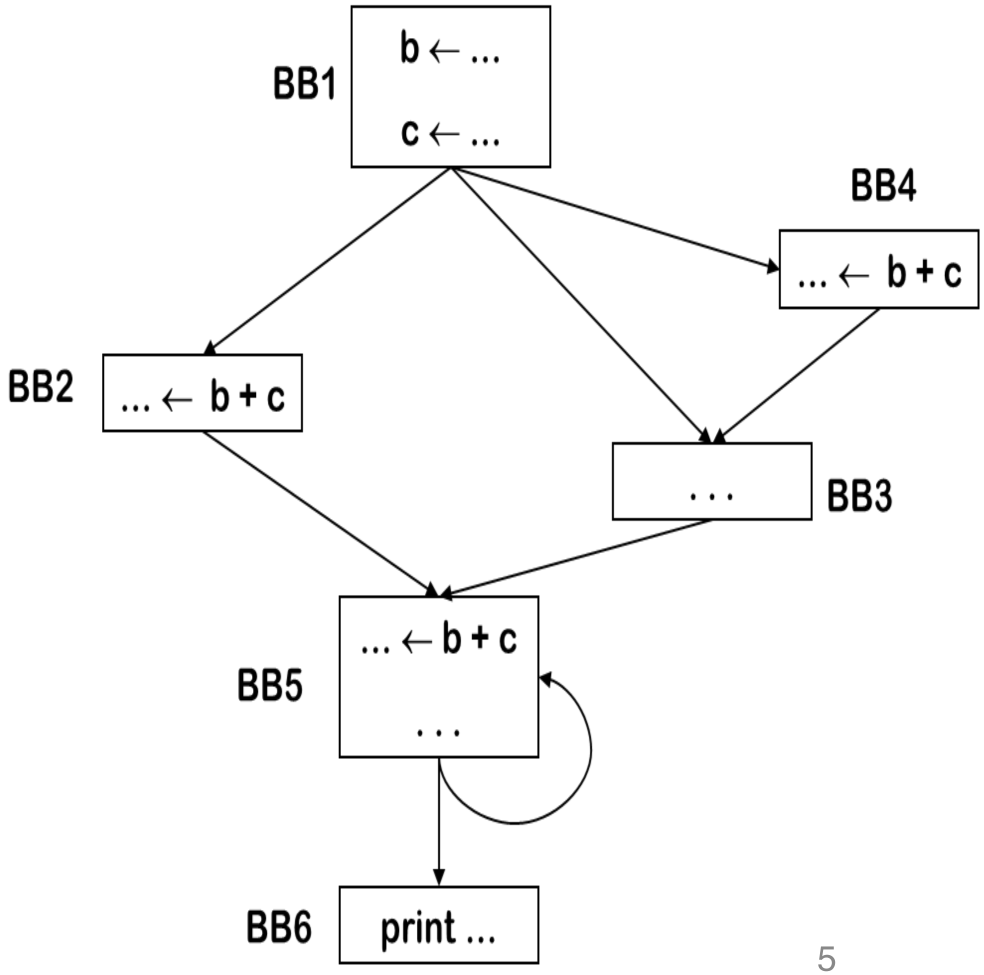
Question1.

Consider the control flow graph shown below. Indicate where computations of **b+c** can be inserted and deleted to minimize the number of times it is computed. Assume that there are no other defs of b and c, and do not worry about dead code elimination in this example.



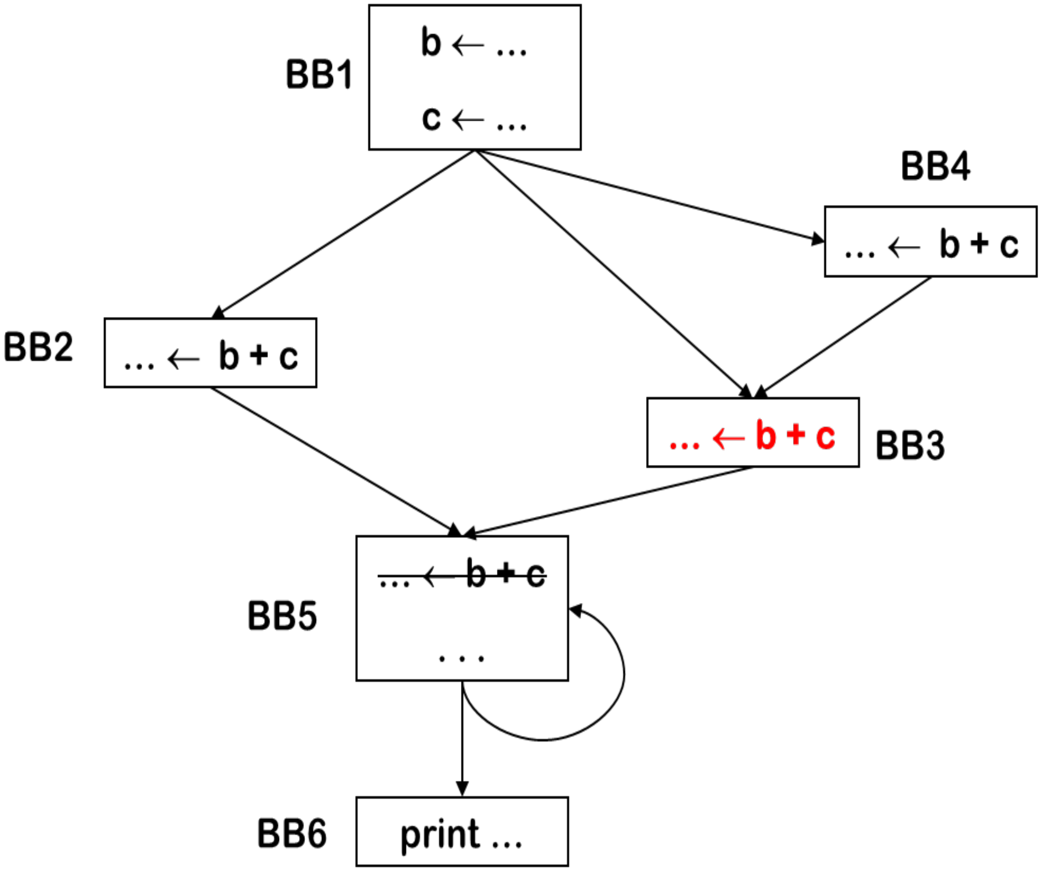
Sample solution : 1st step

- Computation of **b+c** in BB5 is partially redundant.
- We can remove the redundancy by moving the computation of **b+c** from BB5 to a location before BB5.



Sample solution : 2nd step

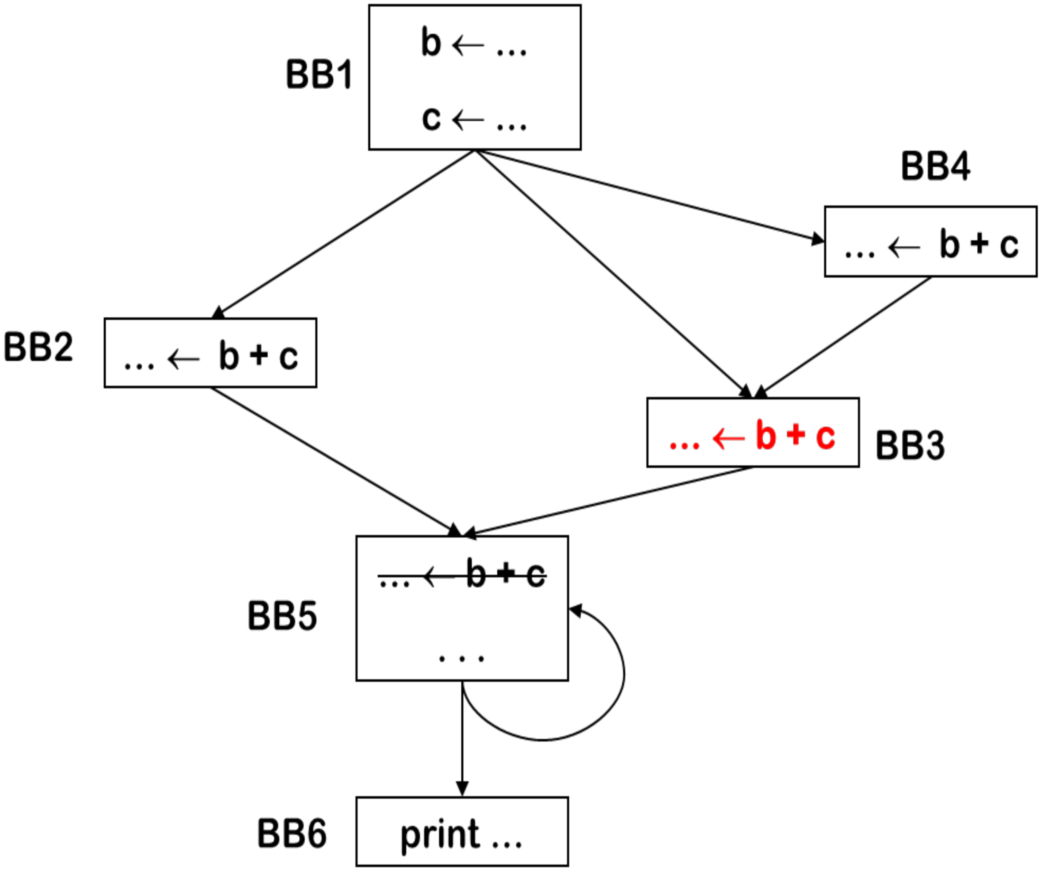
- By moving the computation of b+c to the end of BB3, redundancy of computing b+c along the path **[BB1→BB2→BB5 →* BB6]** is removed.
- Redundancy still remains along the path **[BB1→BB4→BB3→BB5 →* BB6]**.



Sample solution : 2nd step

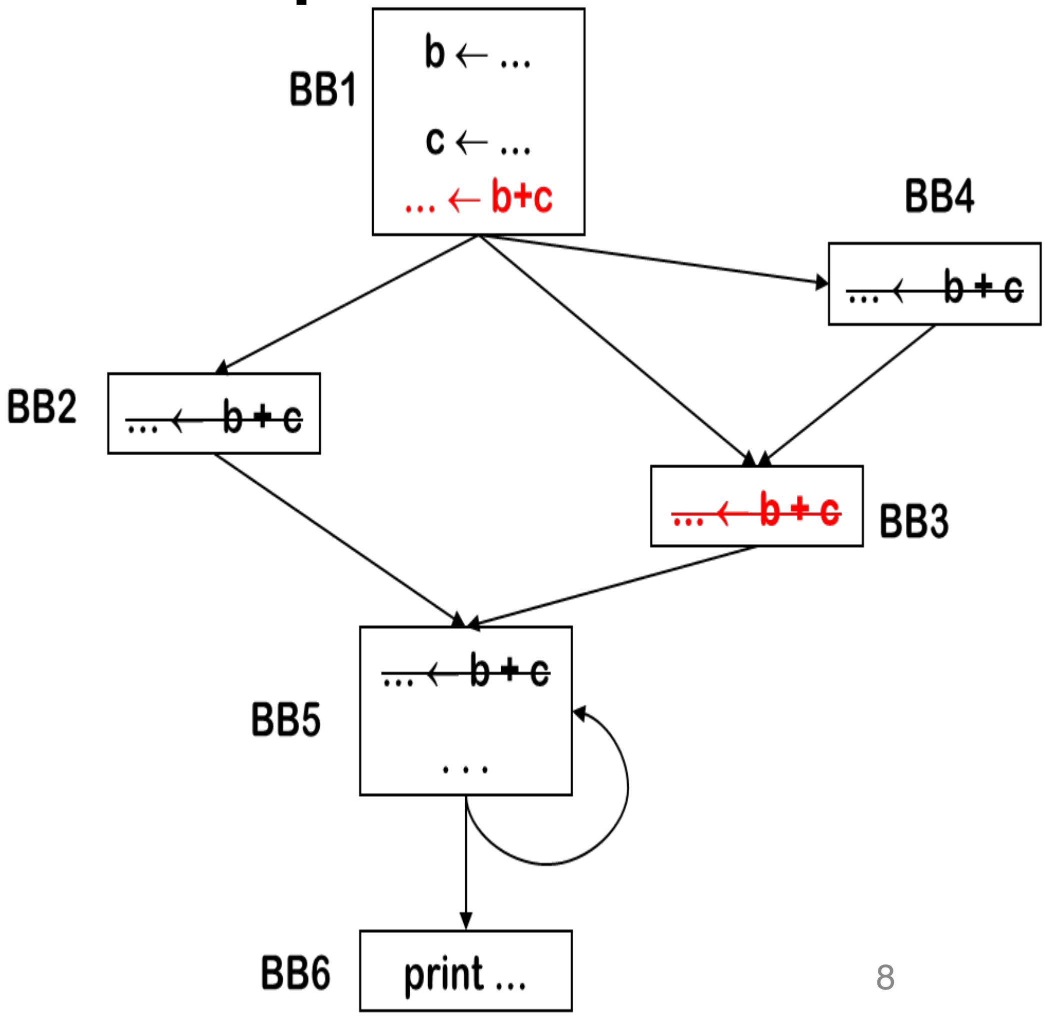
- **b+c** is computed on every path that leaves ***BB1*** and produces the same value at each of those computations.

  (= **b+c** is an anticipable expression from the end of BB1)



Sample solution : 3rd step

- Since **b+c** is anticipable from the end of BB1, it is safe to append the computations of **b+c** to the end of BB1, and delete others.
- After the modification, there are no redundancies remaining in any control path.

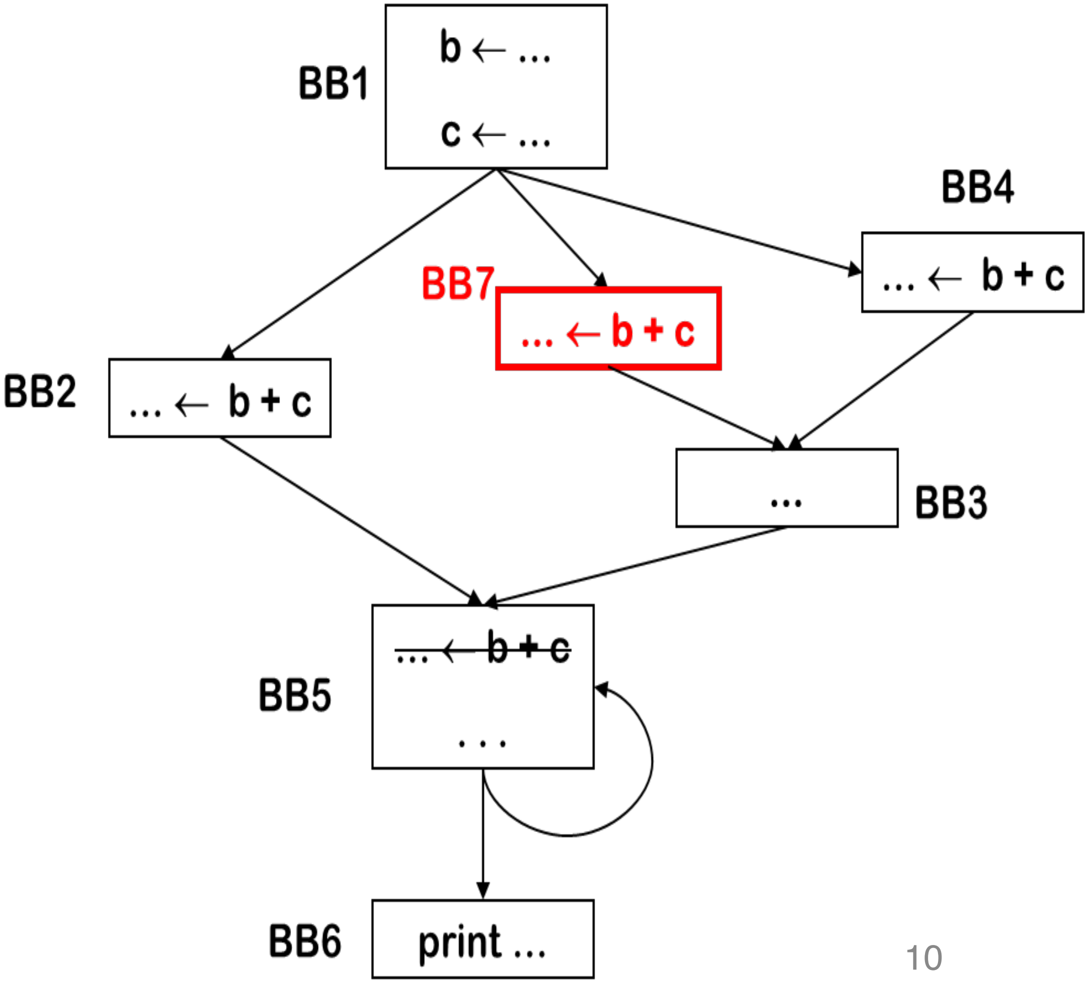


Comments on students’ answers

- Almost all students submitted same answers as the sample solution.
- We will discuss some of the different solutions from students.

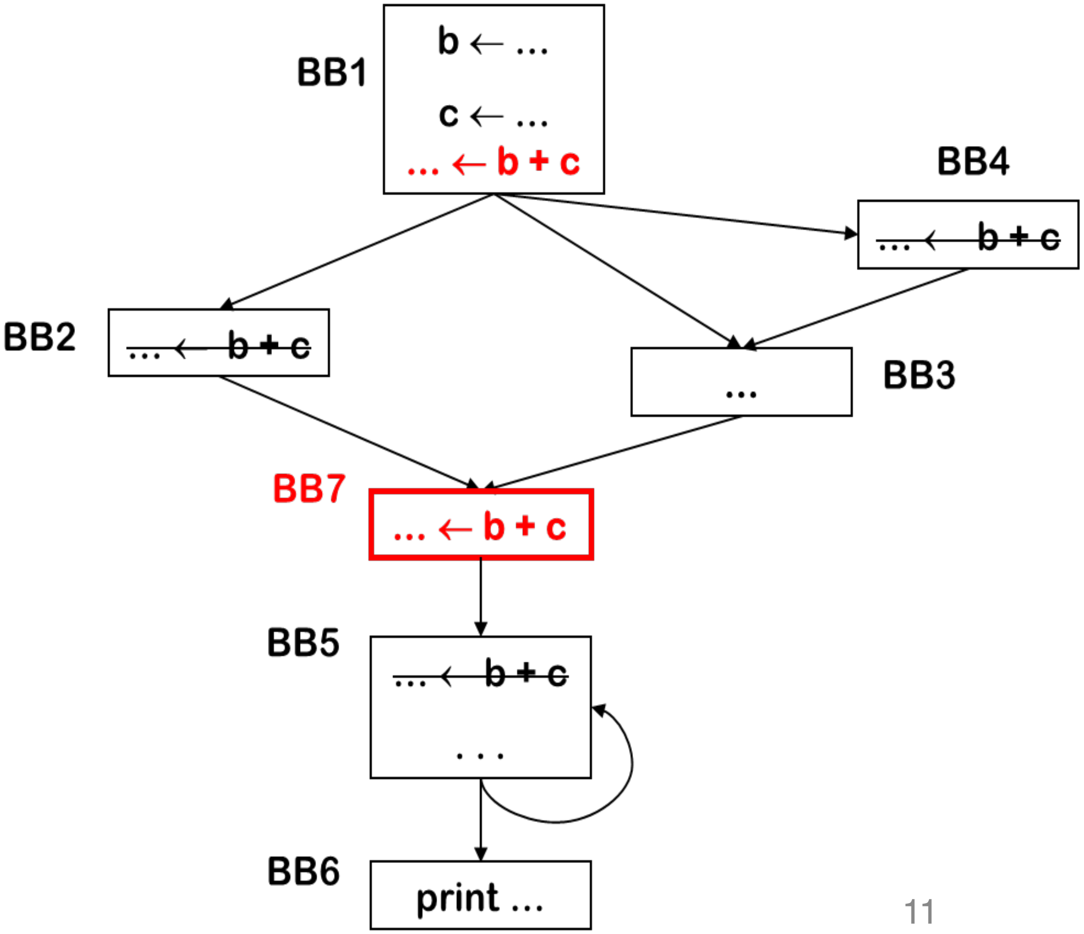
Alternate student solutions #1

- Computation of **b+c** in BB5 is deleted, and a new basic block(BB7) is added with computation of **b+c**.
- **b+c** is computed only once in every control path, so there are no redundancies.
- However, this solution would result in **longer code length** than the sample solution.



Alternate student solutions #2

- A new basic block(BB7) is added before BB5, computation of **b+c** is deleted from BB4 and BB2.
- Adding a new block before BB5 can reduce the redundancy caused by the loop in BB5, but computation of **b+c** is done twice in every control flow.



Where we are in the class so far …

- Lecture 1: Compiler Structure, Intermediate Representations
  - Introduced simple dead code elimination as a motivating example of code optimization
- Lecture 2: Control Flow Graphs, Reaching Definitions
- Lecture 3: Introduction to Data Flow Analysis
  - Introduced reaching definitions as a data flow analysis for improved dead code elimination
- Lecture 4: Value Numbering, Dominators
- Lecture 5: Lazy Code Motion, Available Expression Analysis
  - Introduced redundancy elimination as second motivating example of code optimization
  - Different levels of redundancy elimination algorithms
    - Local Value Numbering (LVN)
    - Superlocal Value Numbering (SVN)
    - Dominator VN Technique (DVNT)
    - Lazy Code Motion (LCM)
    - Available Expression Analysis

Motivation for today's lecture

- Classical data flow algorithms incur a lot of overhead due to propagation of sets of definitions, expressions, etc.
- Algorithms can be made more efficient if each use only had a single definition
  - Single assignment property
  - Motivated by functional programming
- How do we get the single assignment property in imperative code?
- Answer: convert to Static Single Assignment (SSA) form!

Static Single Assignment Form

- The main idea: each name defined exactly once
- Introduce ϕ-functions to make it work

Original

```
x ← …
y ← …
while (x < k)
    x ← x + 1
    y ← y + x
```

SSA-form

```
         $x_0$ ← …
         $y_0$ ← …
         if ($x_0$ >= k) goto next
loop:        $x_1$ ← $\phi(x_0,x_2)$
             $y_1$ ← $\phi(y_0,y_2)$
             $x_2$ ← $x_1$ + 1
             $y_2$ ← $y_1$ + $x_2$
             if ($x_2$ < k) goto loop
next:        …
```

Strengths of SSA-form

- Sharper analysis
- ϕ-functions give hints about placement
- (sometimes) faster algorithms

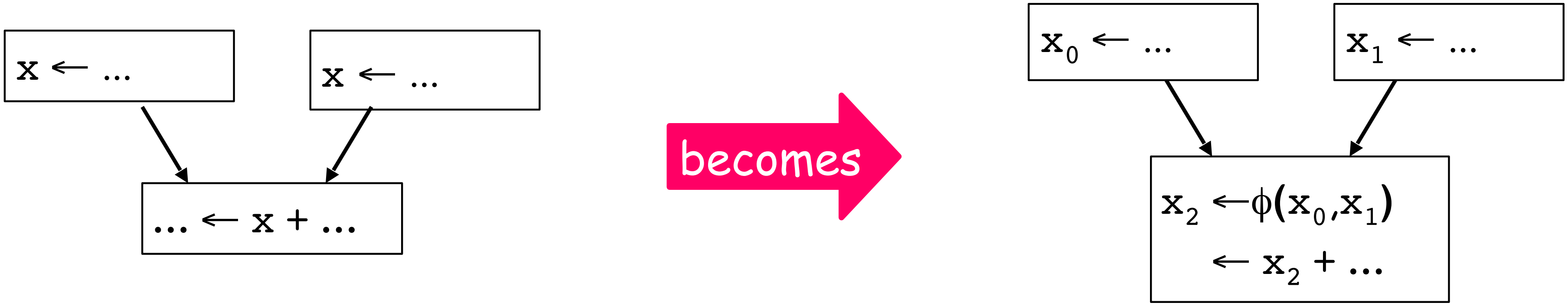
SSA Name Space (in general)

Two principles

- Each name is defined by exactly one operation
- Each operand refers to exactly one definition

To reconcile these principles with real code

- Insert ϕ-functions at merge points to reconcile name space
- Add subscripts to variable names for uniqueness



$x \leftarrow \ldots$

$x \leftarrow \ldots$

$\ldots \leftarrow x + \ldots$

becomes

$x_0 \leftarrow \ldots$

$x_1 \leftarrow \ldots$

$x_2 \leftarrow \phi(x_0,x_1)$

$\leftarrow x_2 + \ldots$

Review

SSA-form

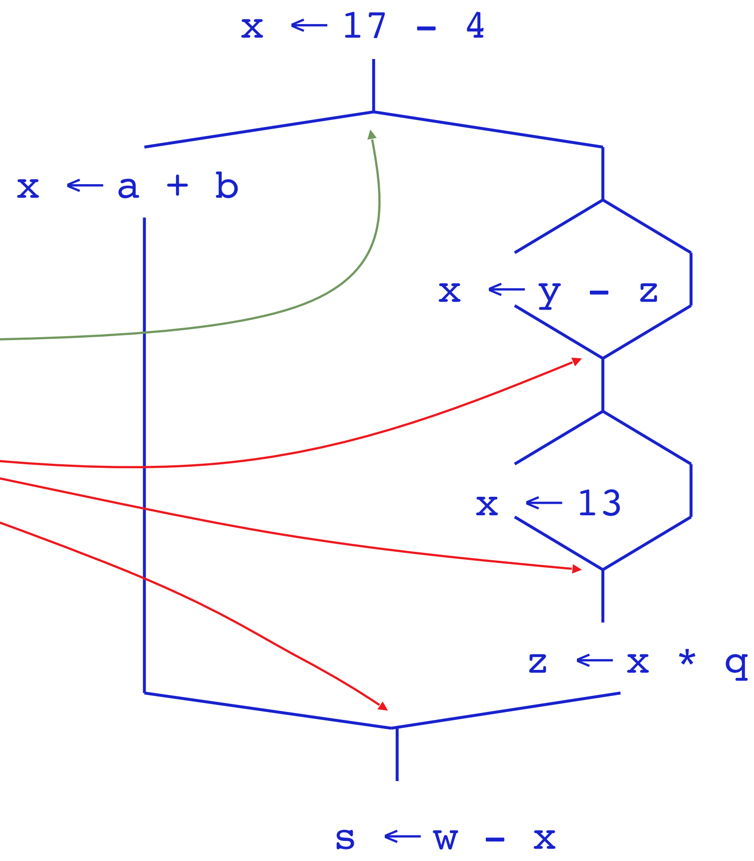
- Each name is defined exactly once
- Each use refers to exactly one name

What's hard

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

Building SSA Form

- Insert Ø-functions at birth points?
- Rename all values for uniqueness



x ← 17 - 4

x ← a + b

x ← y - z

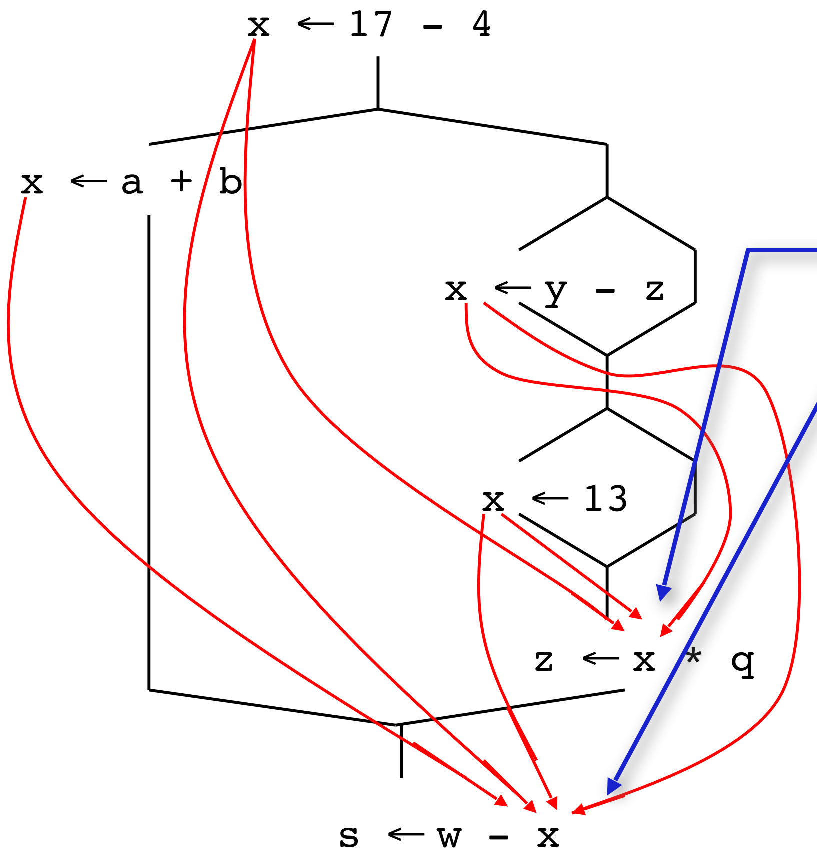
x ← 13

z ← x * q

s ← w - x

Birth Points (another notion due to Tarjan)

Consider the flow of values in this example



The value x appears everywhere

It takes on several values.

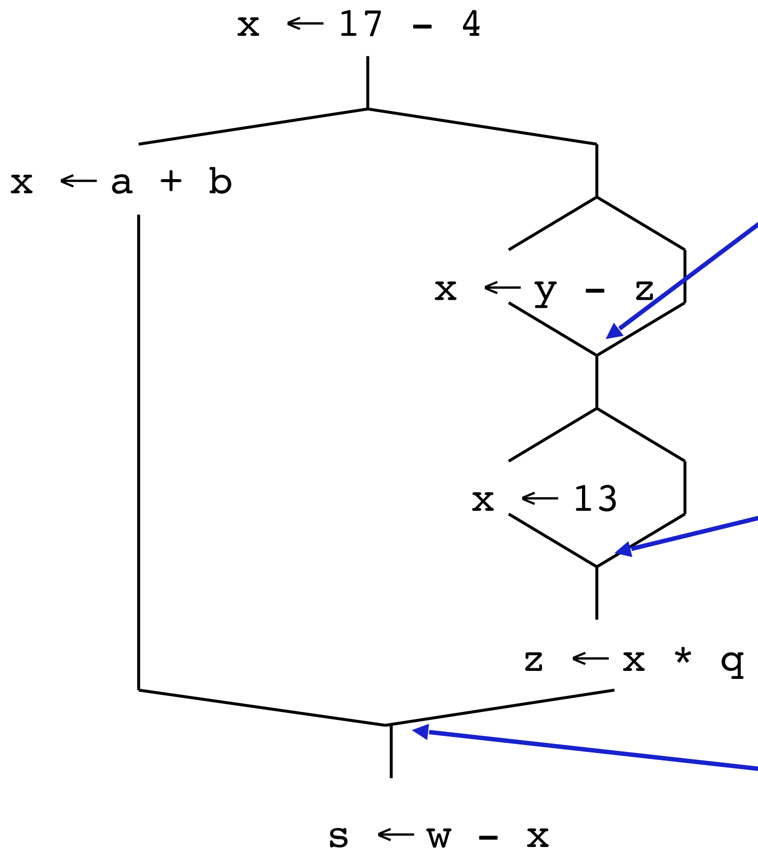
- Here, x can be 13, y-z, or 17-4
- Here, it can also be a+b

If each value has its own name …

- Need a way to merge these distinct values
- Values are "born" at merge points

Birth Points (another notion due to Tarjan)

Consider the flow of values in this example



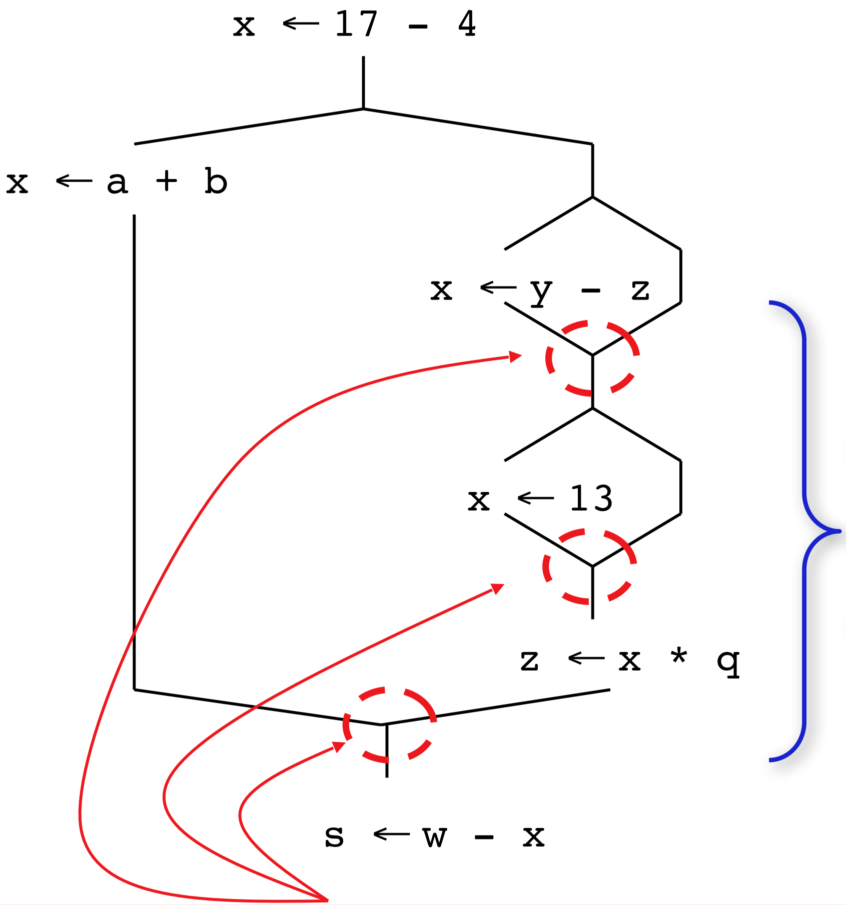
New value for x here
17 - 4 or y - z

New value for x here
13 or (17 - 4 or y - z)

New value for x here
a+b or (13 or (17-4 or y-z))

Birth Points (another notion due to Tarjan)

Consider the flow of values in this example



- All birth points are join points
- Not all join points are birth points
- Birth points are value-specific …

These are all birth points for values

Review

SSA-form

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

What's hard

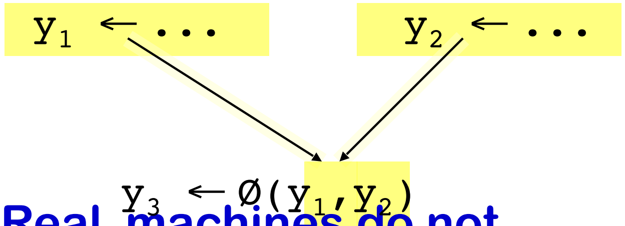
- Straight-line code is trivial
- Splits in the CFG are trivial
- Joins in the CFG are hard

Building SSA Form

- Insert Ø-functions at birth points
- Rename all values for uniqueness

A Ø-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.



Real machines do not implement a Ø-function directly in hardware.

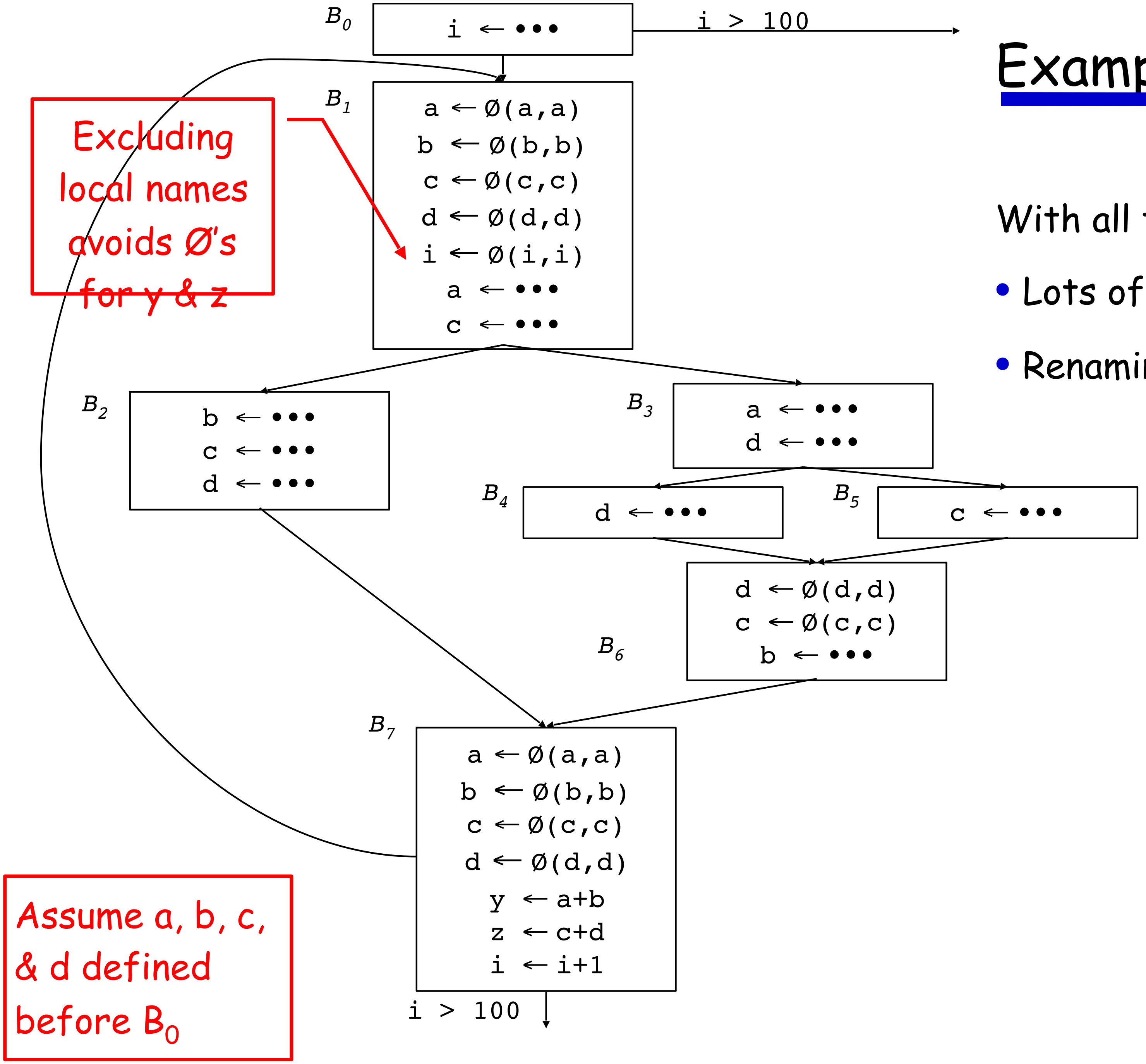
SSA Construction Algorithm (simplified view)

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

Example

With all the Ø-functions

- Lots of new ops
- Renaming is next



B0
```
i ← •••
```
i > 100

B1
```
a ← Ø(a,a)
b ← Ø(b,b)
c ← Ø(c,c)
d ← Ø(d,d)
i ← Ø(i,i)
a ← •••
c ← •••
```

B2
```
b ← •••
c ← •••
d ← •••
```

B3
```
a ← •••
d ← •••
```

B4
```
d ← •••
```

B5
```
c ← •••
```

B6
```
d ← Ø(d,d)
c ← Ø(c,c)
b ← •••
```

B7
```
a ← Ø(a,a)
b ← Ø(b,b)
c ← Ø(c,c)
d ← Ø(d,d)
y ← a+b
z ← c+d
i ← i+1
```
i > 100

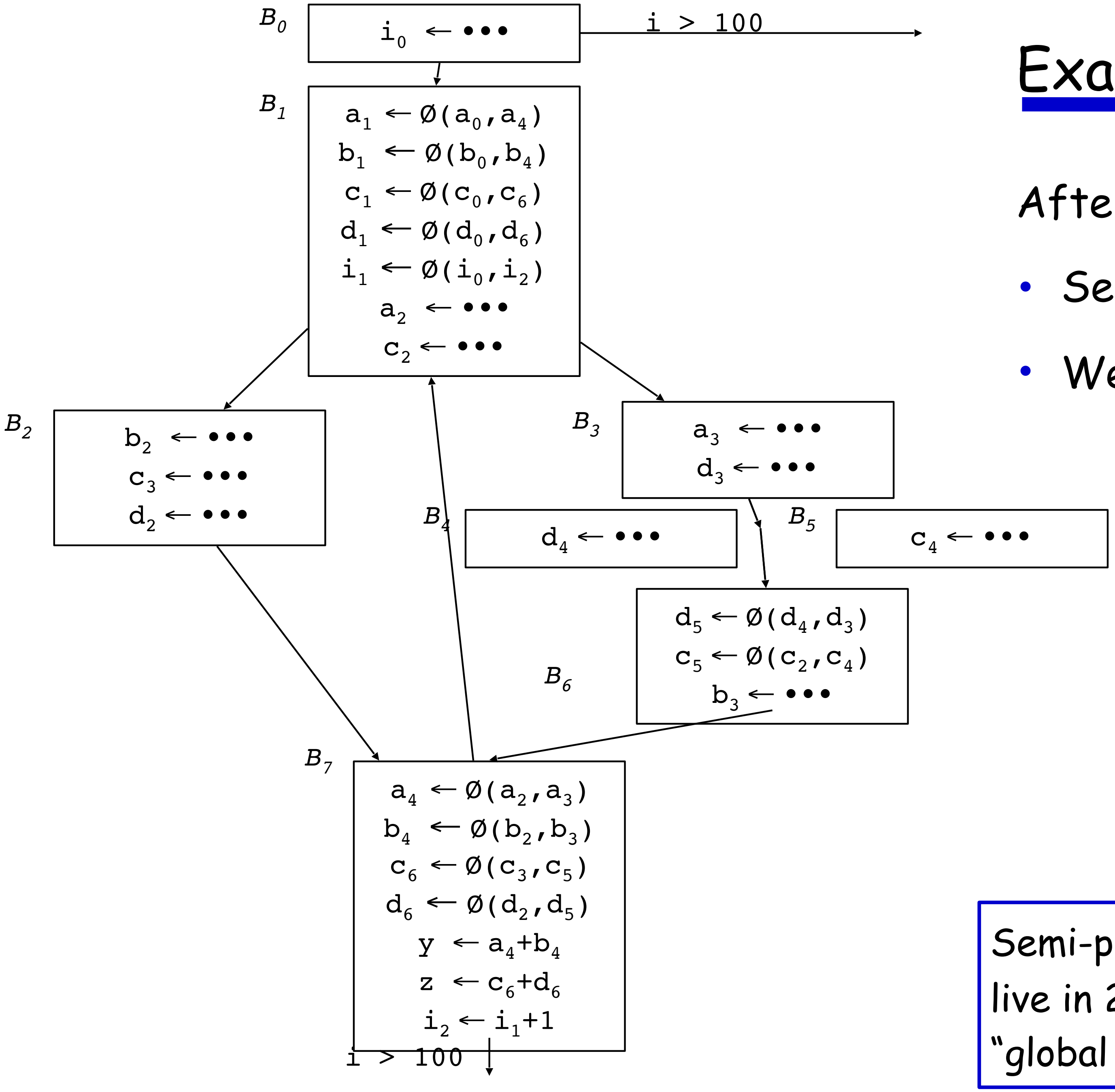
Excluding local names avoids Ø's for y & z

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

Example

After renaming

- Semi-pruned SSA form
- We're done …



B_0

$i_0 \leftarrow$ •••

i > 100

B_1

$a_1 \leftarrow \emptyset(a_0, a_4)$

$b_1 \leftarrow \emptyset(b_0, b_4)$

$c_1 \leftarrow \emptyset(c_0, c_6)$

$d_1 \leftarrow \emptyset(d_0, d_6)$

$i_1 \leftarrow \emptyset(i_0, i_2)$

$a_2 \leftarrow$ •••

$c_2 \leftarrow$ •••

B_2

$b_2 \leftarrow$ •••

$c_3 \leftarrow$ •••

$d_2 \leftarrow$ •••

B_3

$a_3 \leftarrow$ •••

$d_3 \leftarrow$ •••

B_4

$d_4 \leftarrow$ •••

B_5

$c_4 \leftarrow$ •••

B_6

$d_5 \leftarrow \emptyset(d_4, d_3)$

$c_5 \leftarrow \emptyset(c_2, c_4)$

$b_3 \leftarrow$ •••

B_7

$a_4 \leftarrow \emptyset(a_2, a_3)$

$b_4 \leftarrow \emptyset(b_2, b_3)$

$c_6 \leftarrow \emptyset(c_3, c_5)$

$d_6 \leftarrow \emptyset(d_2, d_5)$

$y \leftarrow a_4 + b_4$

$z \leftarrow c_6 + d_6$

$i_2 \leftarrow i_1 + 1$

i > 100

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 Ø-functions
- Inserts some Ø-functions where they are dead
- Would like to avoid inserting them

Two ideas

- Semi-pruned SSA: discard names used in only one block
  - Significant reduction in total number of Ø-functions
  - Needs only local Live 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
  - Requires global Live variable analysis (more expensive)

In practice, both are simple modifications to SSA construction

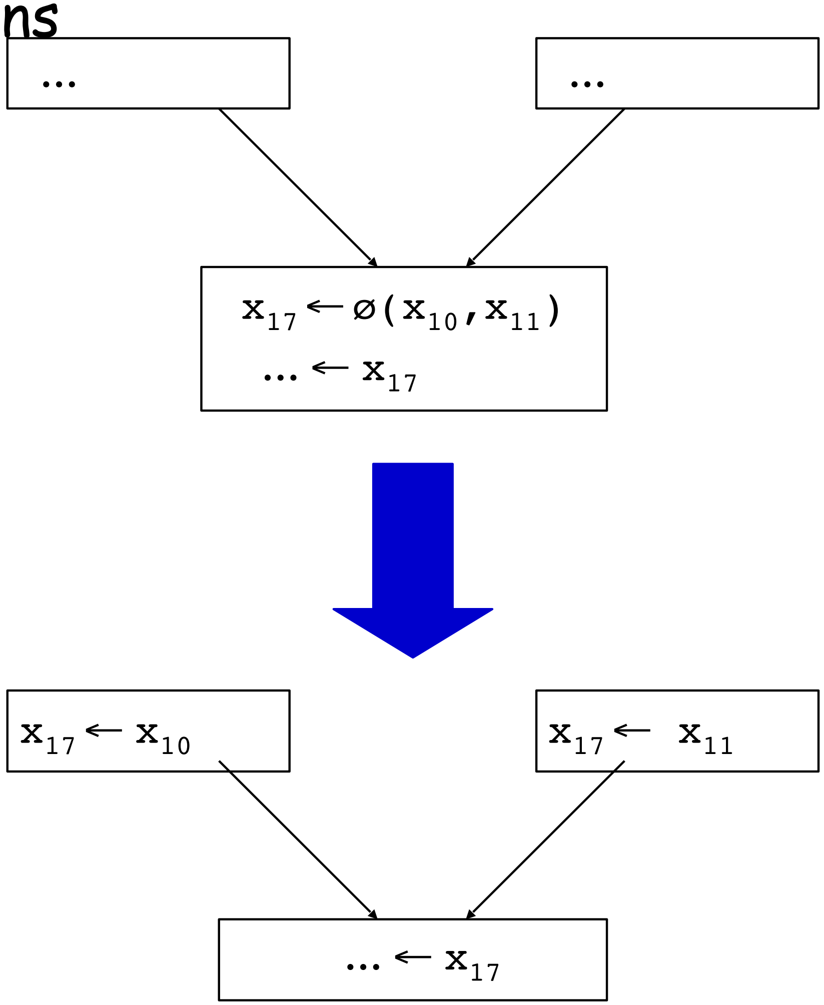
SSA Deconstruction

At some point, we need executable code

- Few machines implement Ø operations
- Need to fix up the flow of values

Basic idea

- Insert copies Ø-function pred's
- Simple algorithm
  - Works in most cases
- Adds lots of copies
  - Most of them coalesce away



…

…

$x_{17} \leftarrow ø(x_{10}, x_{11})$

$… \leftarrow x_{17}$

$x_{17} \leftarrow x_{10}$

$x_{17} \leftarrow x_{11}$

$… \leftarrow x_{17}$