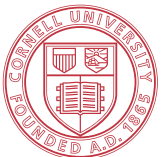


ECE 6775
High-Level Digital Design Automation
Fall 2024

More CFG
Static Single Assignment



Cornell University



Announcements

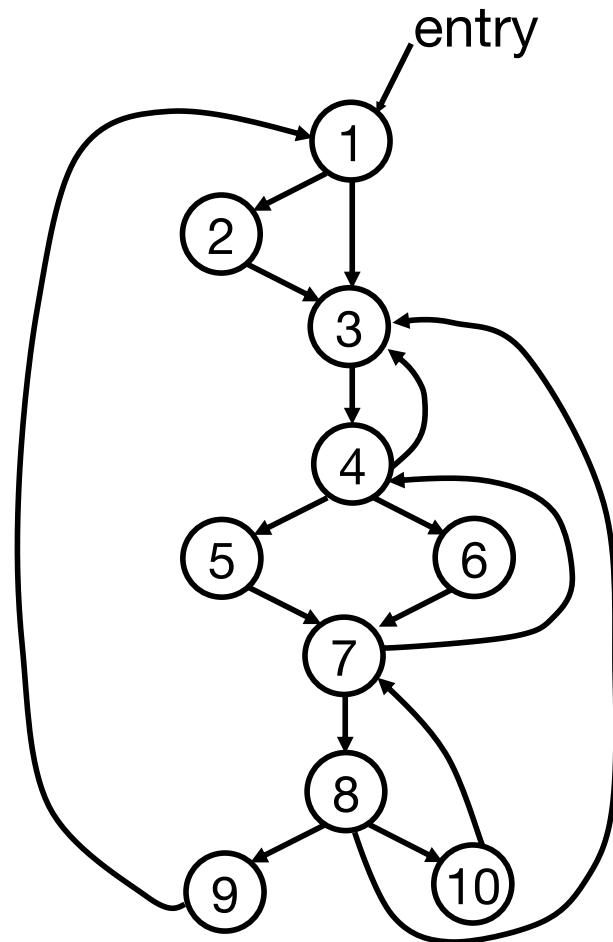
- ▶ **In-class midterm moved to Oct 22**
- ▶ Lecture on Oct 1 may be rescheduled, date TBD
(a poll will be posted on Ed)

Agenda

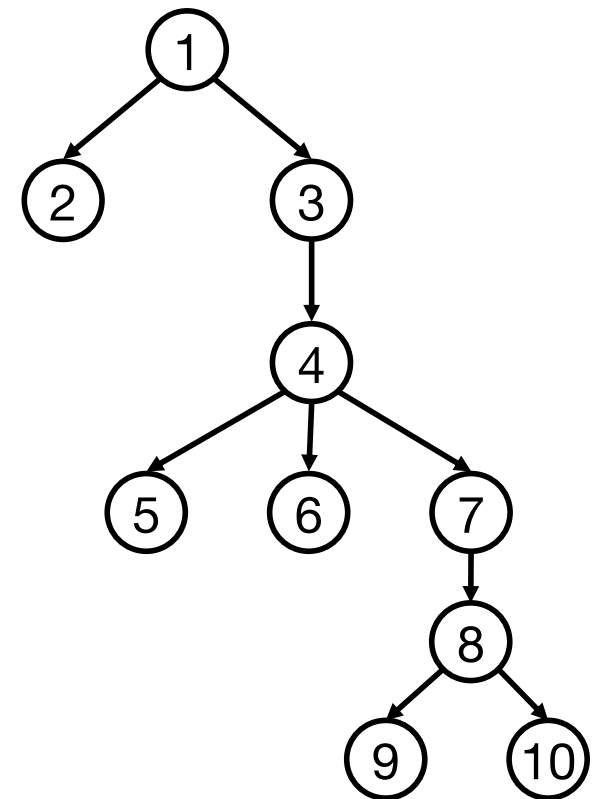
- ▶ More control flow analysis
 - Dominator tree
 - Dominance frontier
- ▶ Data flow graph (DFG) and data dependences
- ▶ Static single assignment (SSA)
 - SSA definition
 - PHI node (Φ -node) placement
 - Code optimizations with SSA

Review: Dominator Tree

CFG



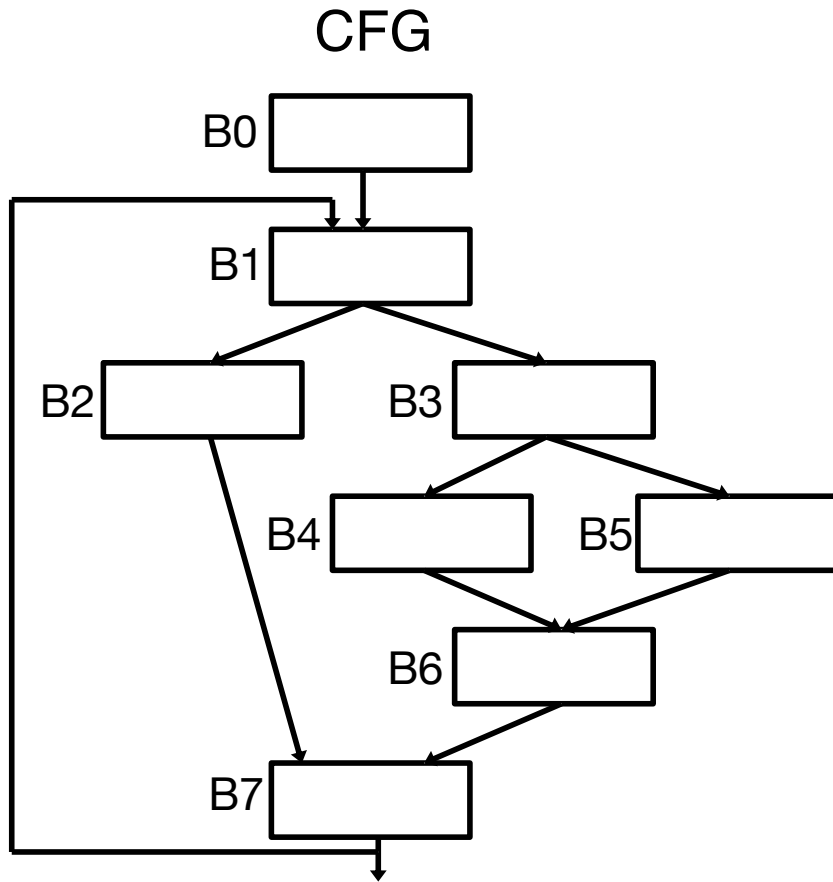
Dominator Tree



Dominance Frontier

- ▶ A basic block q is in the **dominance frontier set (DF)** of basic block p if and only if
 - (1) p *does NOT strictly dominate* q
 - (2) p *dominates some predecessor(s)* of qIf above two conditions hold, $q \in \text{DF}(p)$
- ▶ Informally, for an $q \in \text{DF}(p)$, q is almost strictly dominated by p
- ▶ Useful for efficiently computing the SSA form

Example: Dominance Frontiers



Dominance
frontiers (DF)

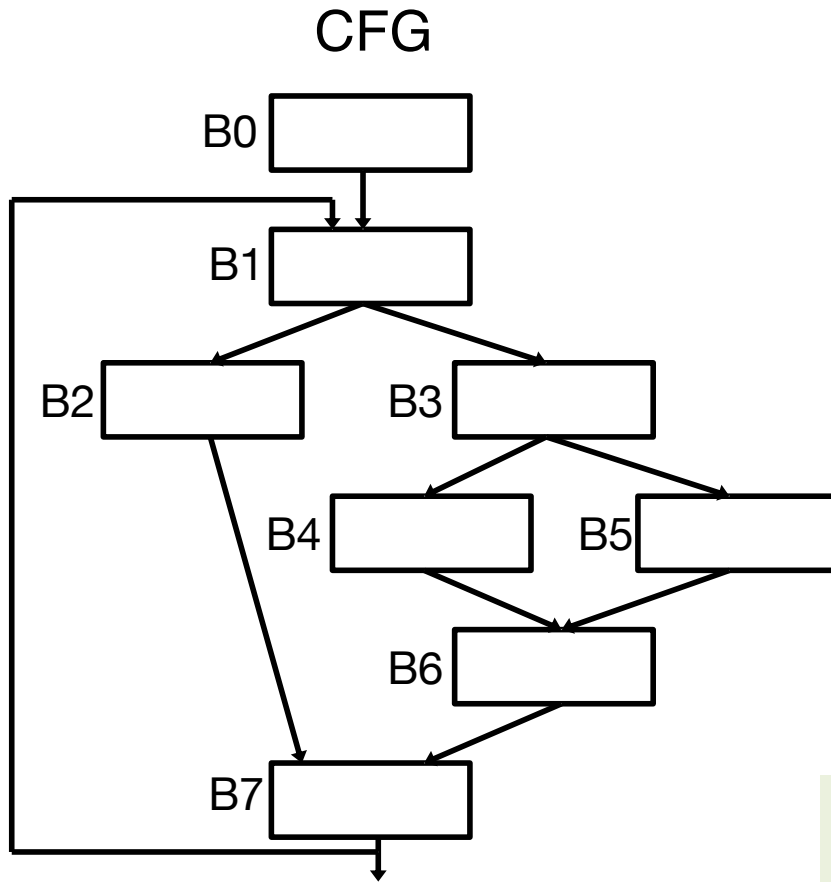
p	DF(p)
0	—
1	?
2	7
3	7
4	6
5	6
6	7
7	?

q is in the dominance frontier set (DF) of p iff

- (1) p does NOT strictly dominate q
- (2) p dominates some predecessor(s) of q

If above two conditions hold, $q \in DF(p)$

Dominance Frontiers and Dominator Tree

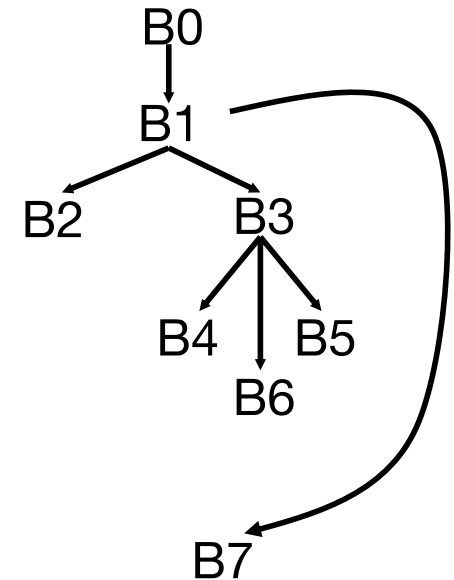


A **convergence point** is a node in CFG with multiple predecessors (B1, B6, B7 in this example)

Dominance frontiers (DF)

p	DF(p)
0	—
1	1
2	7
3	7
4	6
5	6
6	7
7	1

Dominator tree



```

/* Algorithm to construct the DF sets */
foreach convergence point Q in CFG
  foreach predecessor X of Q in CFG
    Run up to Y=IDOM(Q) in the dominator tree;
    Add Q to DF(P) for each P between [X, Y)
  
```

Only convergence points appear on the DF sets!

Data Flow Graph

- ▶ A **data flow graph (DFG)** represents the flow of data between operations in a program
 - **Nodes:** represent operations, e.g., arithmetic operations, memory accesses
 - **Edges:** represent the flow of data between these operations
- ▶ DFG captures **data dependences** between computations, which is crucial for compilers to perform optimizations (e.g., recording, parallelization) while ensuring correctness

Data Dependences

► Types of **data dependences**

- True dependences, anti-dependences, output dependences
- Intra-iteration (loop independent), inter-iteration (loop carried)
 - *In this lecture, we focus on intra-iteration dependences on scalars*

► **True dependence**

- Also known as Read After Write (RAW) or flow dependence
- $S1 \rightarrow^t S2$: S1 precedes S2 in the program execution and computes a value that S2 uses

► **Anti-dependence**

- Also known as Write After Read (WAR) dependence
- $S1 \rightarrow^a S2$: S1 precedes S2 in the program execution and may read from a variable (or memory location) that is later updated by S2

► **Output dependence**

- Also known as Write After Write (WAW) dependence
- $S1 \rightarrow^o S2$: S1 precedes S2 in the program execution and may write to a variable (or memory location) that is later (over)written by S2

A Simple Example

S1: $x = \text{read}()$

S2: $x = x + 1$

S3: $x = x * 5$

- True dependence (RAW) : $S1 \rightarrow^t S2$; $S2 \rightarrow^t S3$
- Anti dependence (WAR) : $S2 \rightarrow^a S3$
- Output dependence (WAW) : $S1 \rightarrow^o S2$, $S1 \rightarrow^o S3$, $S2 \rightarrow^o S3$

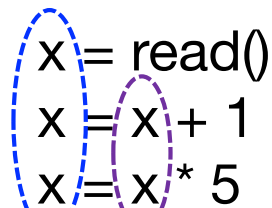
Static Single Assignment

- ▶ The **static single assignment (SSA) form** is a restricted IR where
 - Each (scalar) variable definition has a **unique** name
 - Each (scalar) variable use refers to a single definition
- ▶ SSA simplifies both dataflow and dependence analyses, enabling more effective and streamlined compiler optimizations
 - SSA eliminates artificial dependences (e.g., WAW, WAR) on scalars

SSA within a Basic Block

- ▶ **Assign each variable definition a unique name**
- ▶ Update the uses accordingly

Original code

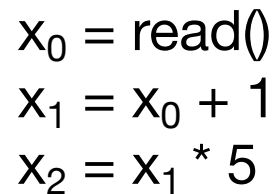


$x = \text{read}()$
 $x = x + 1$
 $x = x * 5$

The original code consists of three lines. The first line, $x = \text{read}()$, is enclosed in a blue dashed oval. The second and third lines, $x = x + 1$ and $x = x * 5$, are enclosed in a purple dashed oval. A blue arrow points from the text 'Defs of x' to the blue oval. A purple arrow points from the text 'Uses of x' to the purple oval.

Defs of x Uses of x

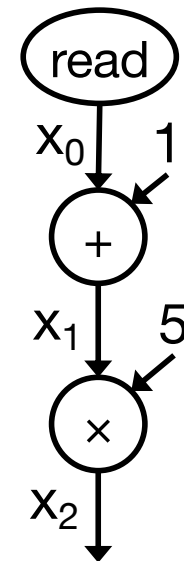
SSA form



$x_0 = \text{read}()$
 $x_1 = x_0 + 1$
 $x_2 = x_1 * 5$

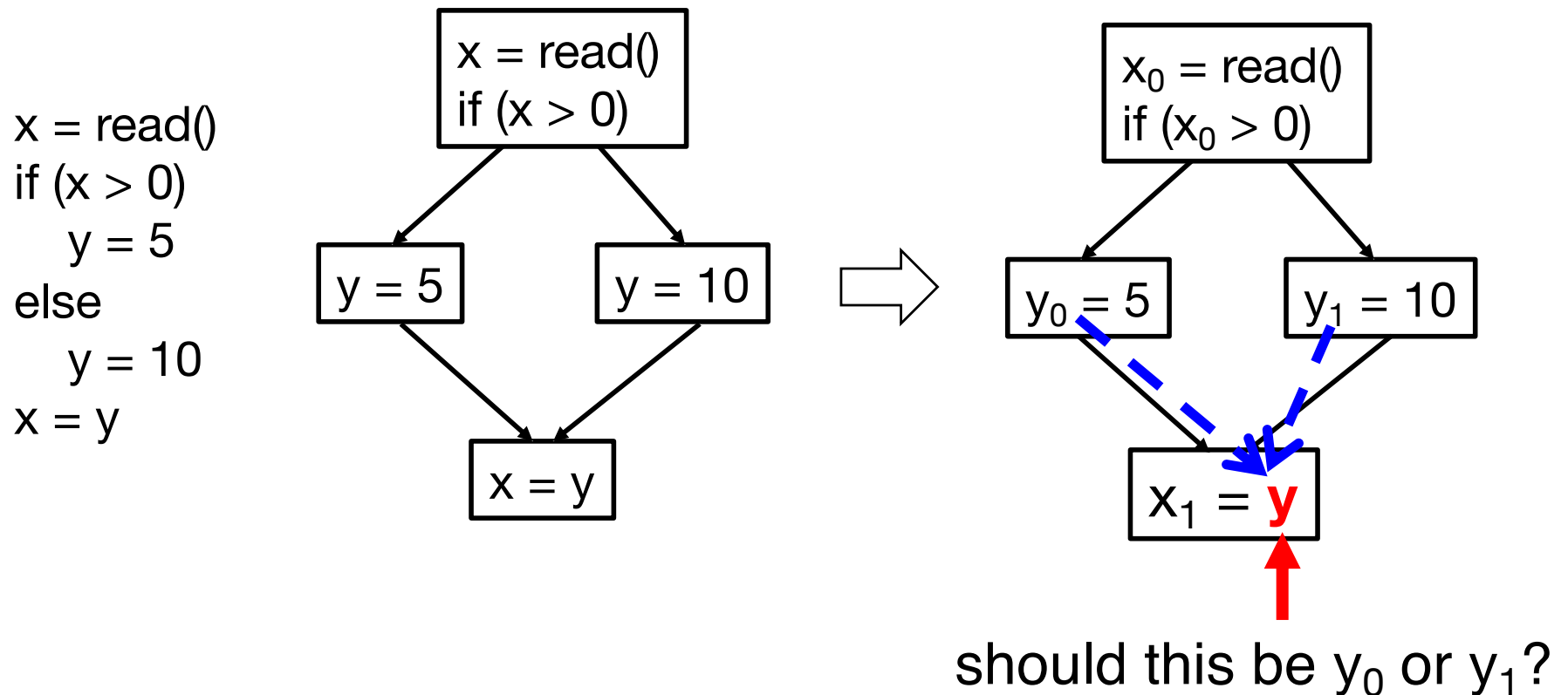
The SSA form code shows three lines. The first line is $x_0 = \text{read}()$. The second line is $x_1 = x_0 + 1$. The third line is $x_2 = x_1 * 5$. A large white arrow points from the original code to the SSA form.

Corresponding
data flow graph



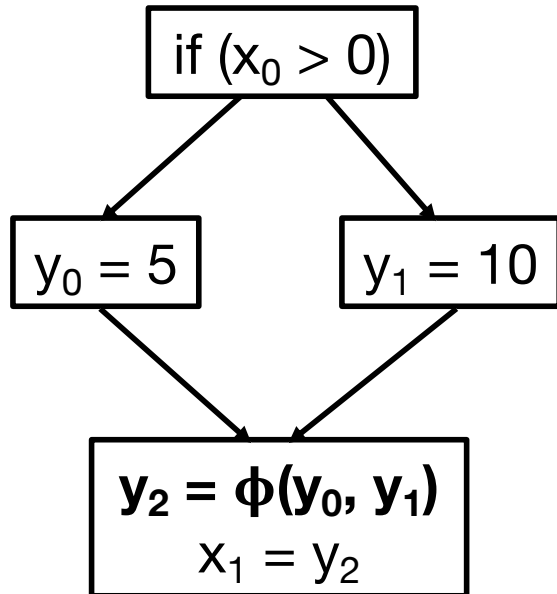
SSA with Control Flow

- Consider a situation where two control-flow paths merge
 - e.g., due to an if-then-else statement or a loop



Introducing ϕ -Node

- Inserts special join functions (called **ϕ -nodes** or PHI nodes) at points where different control flow paths converge



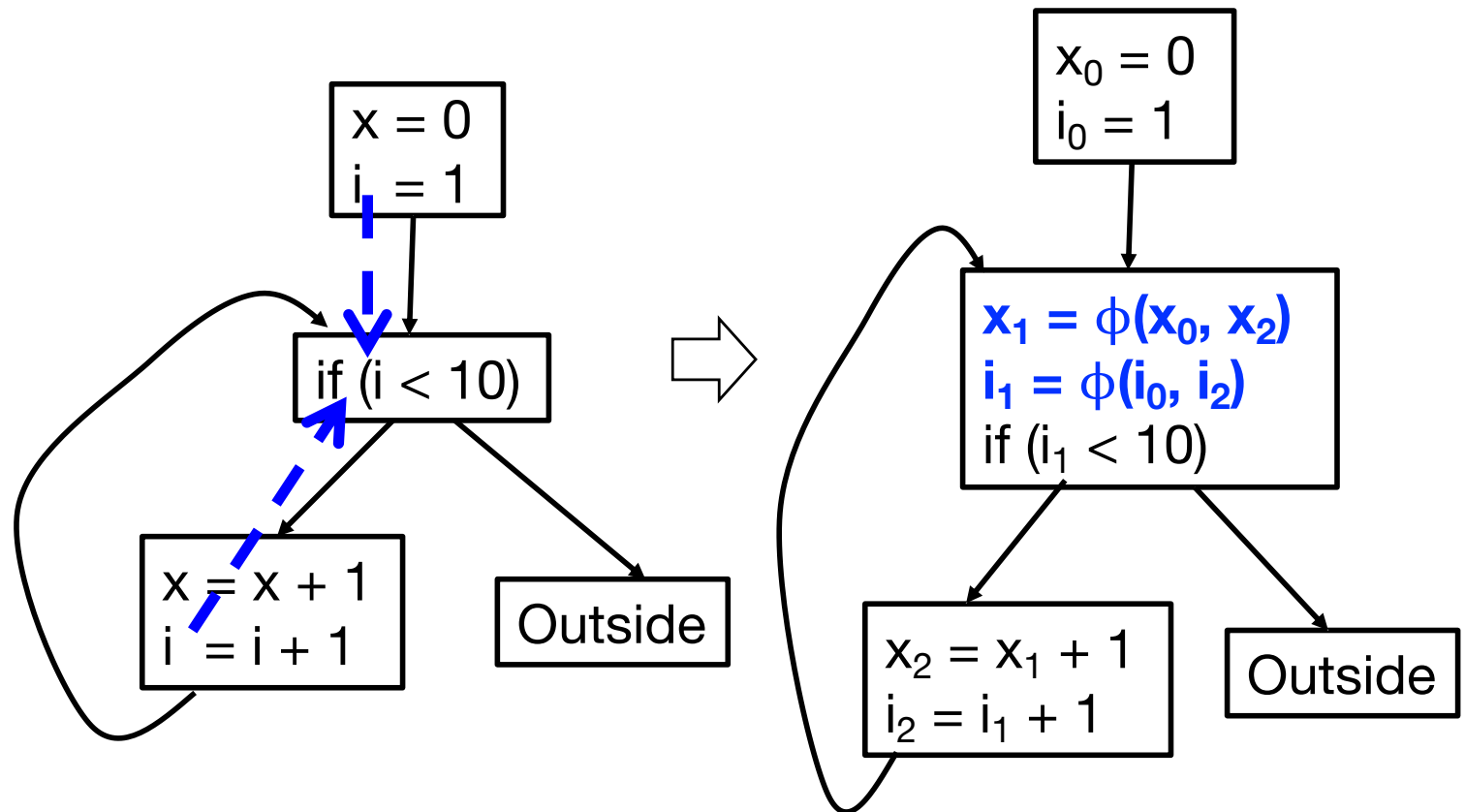
Note: ϕ is not an executable function!

To generate executable code from this form, appropriate copy statements need to be generated in the predecessors (in other words, reversing the SSA process for code generation)

SSA in a Loop

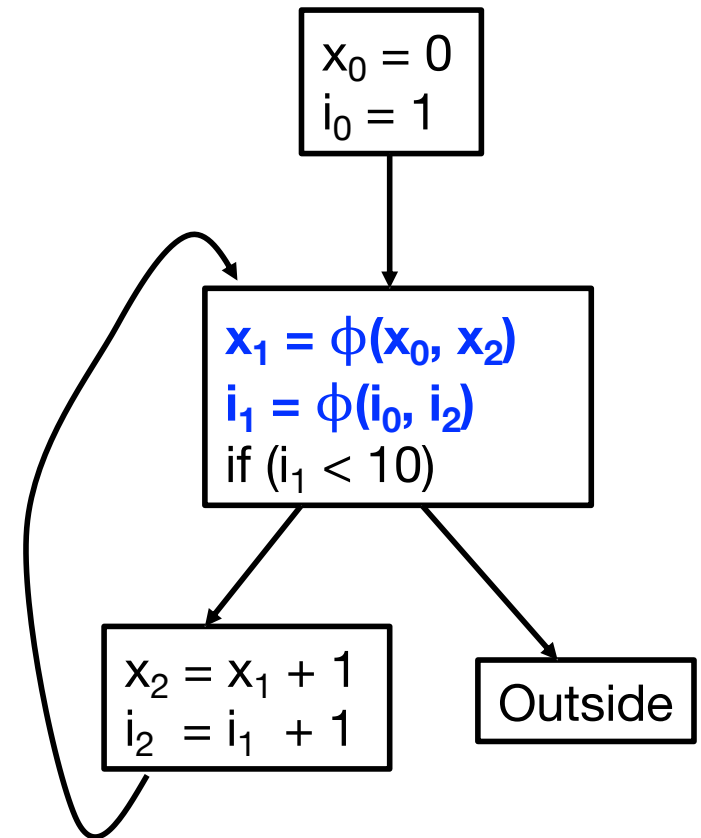
- Insert ϕ -nodes in the loop header block

```
x = 0
i = 1
while (i < 10) {
    x = x + i
    i = i + 1
}
```

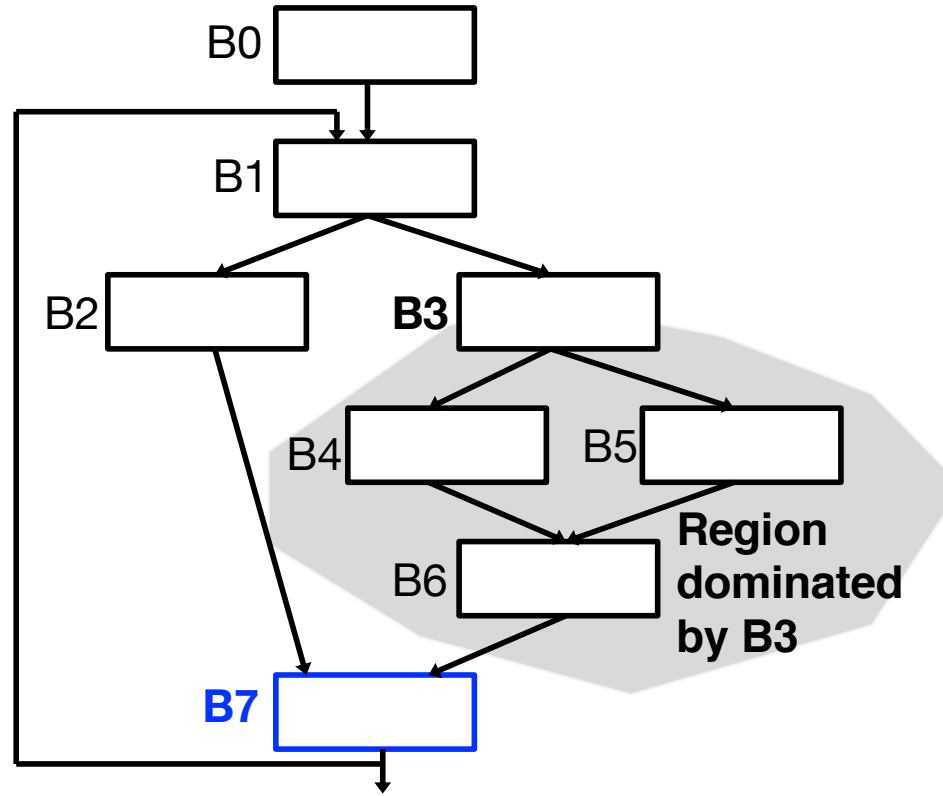


ϕ -Node Placement

- ▶ When and where to insert ϕ -nodes?
 - If two control paths $A \rightarrow C$ and $B \rightarrow C$ converge at a node C , and both A and B contain assignments to variable “ x ”, then ϕ -node for “ x ” must be placed at C
 - **We call C a join node or convergence point**
 - Generalizes to more than two converging control paths
- ▶ Objective: Minimize the number of ϕ -nodes
 - Need to compute dominance frontier sets



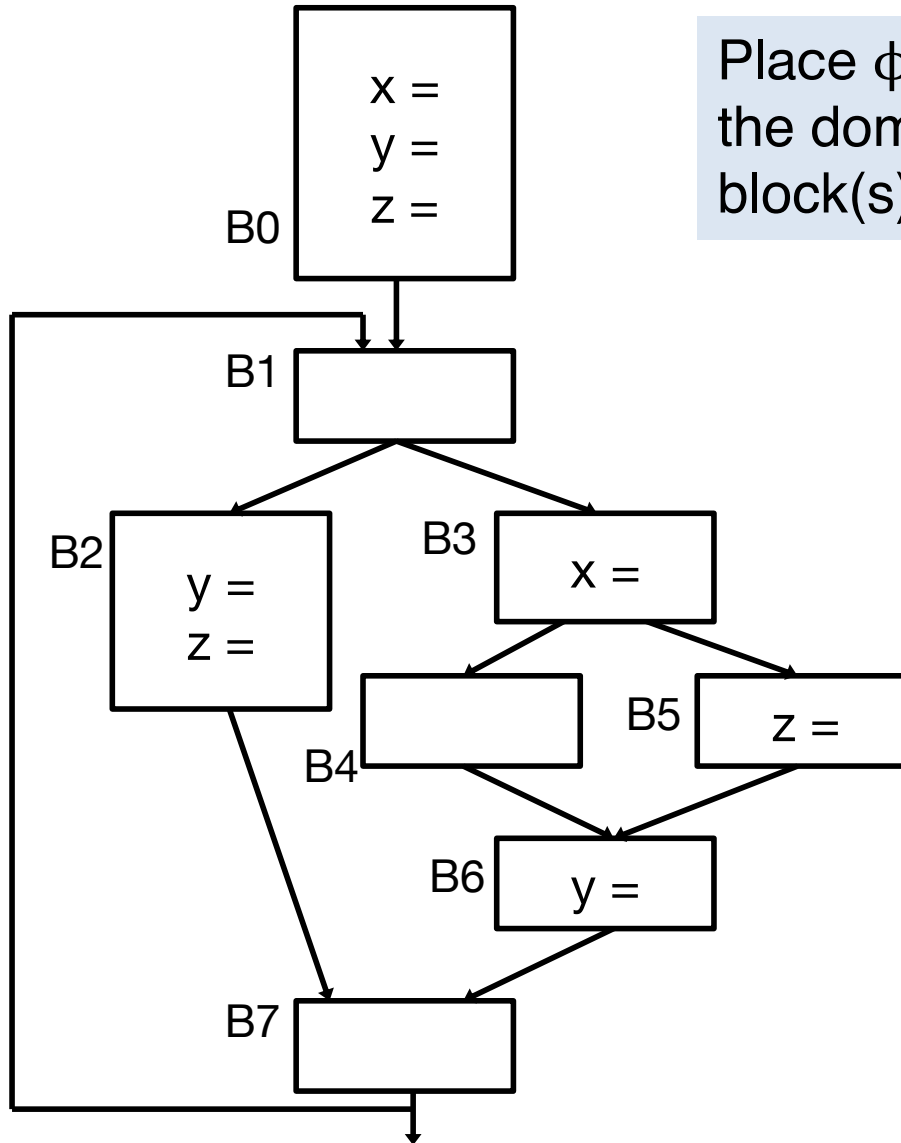
Example: Dominance Frontier and SSA



- ▶ B7 is in the dominance frontier set of B3
 - In other words, B7 is the destination of some edge(s) leaving a region dominated by B3
- ▶ **For each variable definition in B3, a ϕ node is needed in B7**

ϕ -Node Placement

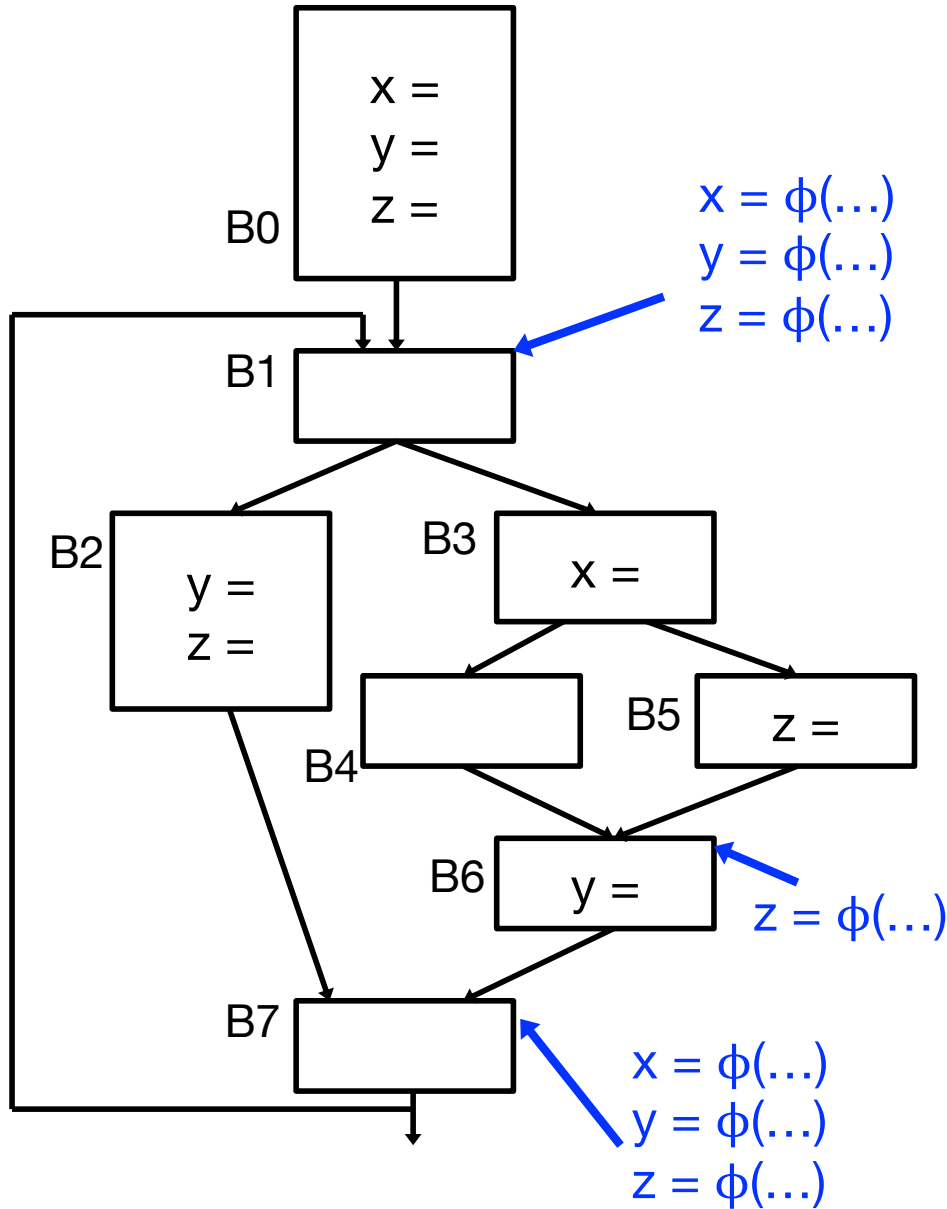
p	DF
0	-
1	1
2	7
3	7
4	6
5	6
6	7
7	1



Place ϕ node(s) of a variable x in the dominance frontier set of the block(s) where x gets defined

ϕ -Node Placement: Iterative Insertion

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



- **x** is defined in 0, 3
=> insert ϕ in 7,
then **x** also defined in 7
=> insert ϕ in 1
- **y** is defined in 0, 2, 6
=> insert ϕ in 7
then **y** also defined in 7
=> insert ϕ in 1
- **z** is defined in 0,2,5
=> insert ϕ in 6,7
then **z** also defined in 7
=> insert ϕ in 1

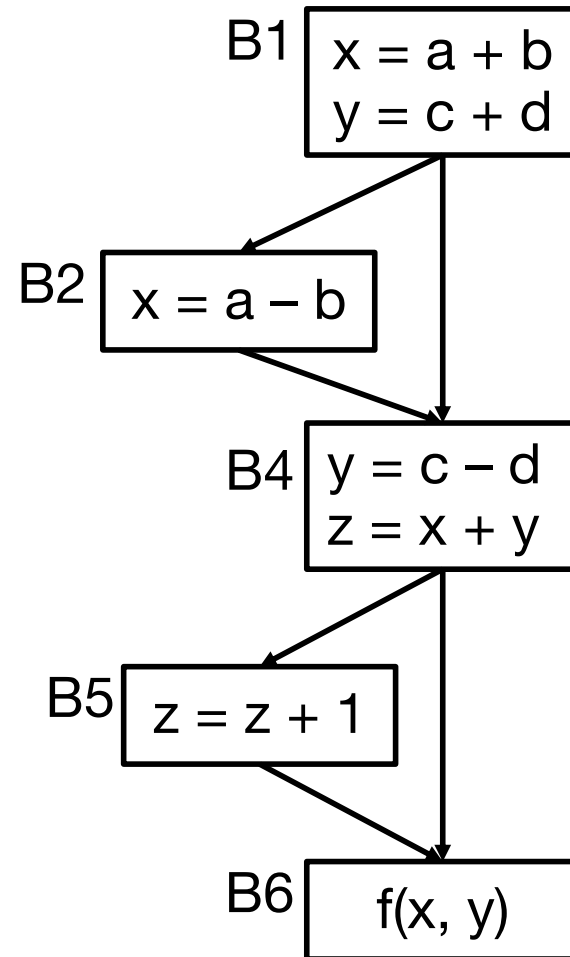
Afterwards, assign a unique name to each variable definition (including ϕ nodes) and update all uses

SSA Applications

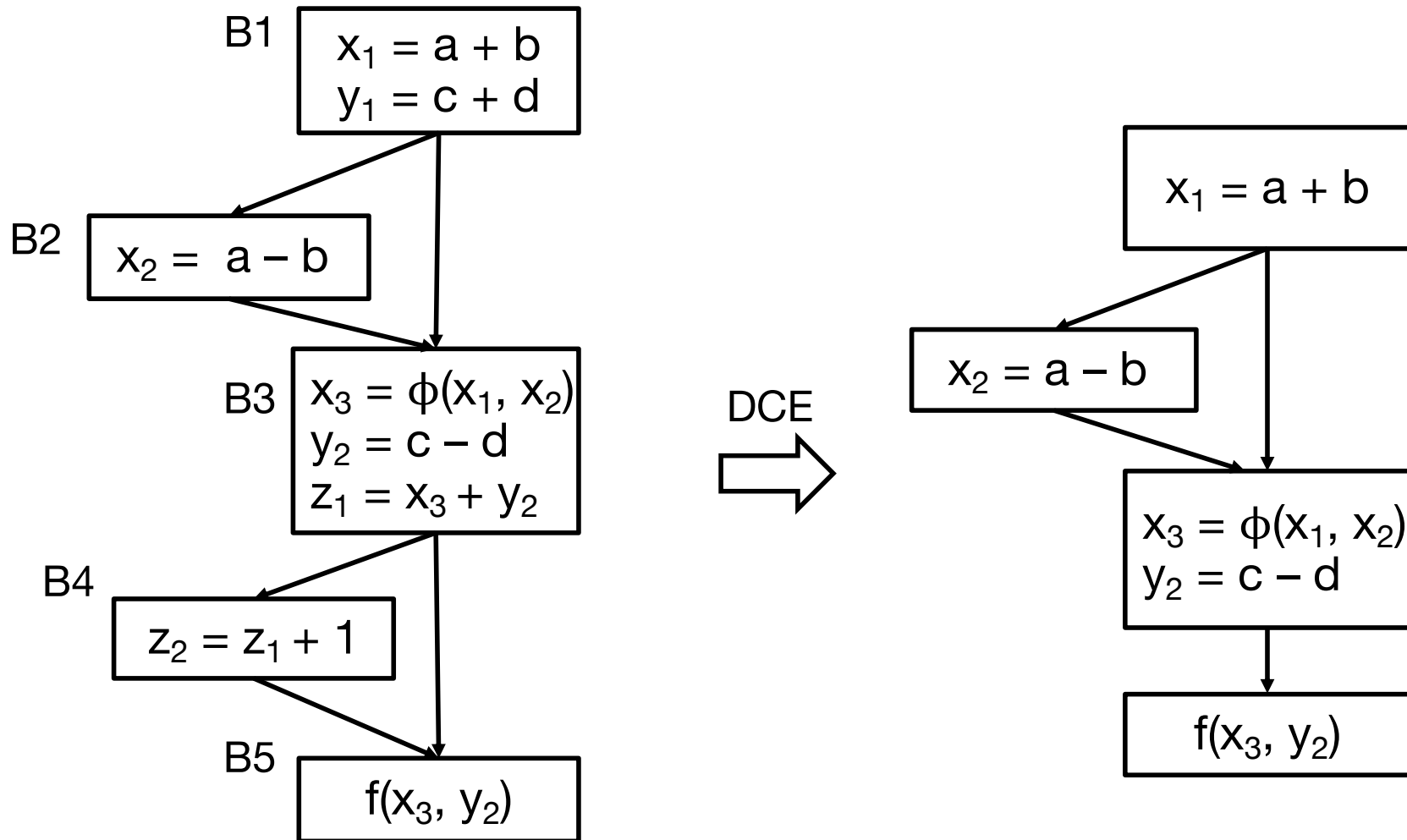
- ▶ SSA form simplifies data flow analysis and many code transformations
 - Primarily due to explicit & simplified (sparse) def-use chains
- ▶ Here we show two simple examples
 - Dead code elimination
 - Loop induction variable detection

Dead Code in CDFG

- ▶ A dead statement is either
 - (1) Unreachable code
 - (2) Definitions never used
- ▶ How to efficiently Identify the dead statements?



Dead Code Elimination (DCE) with SSA



Iteratively remove unused definitions
first remove y_1 , z_2 , and B4; then remove z_1

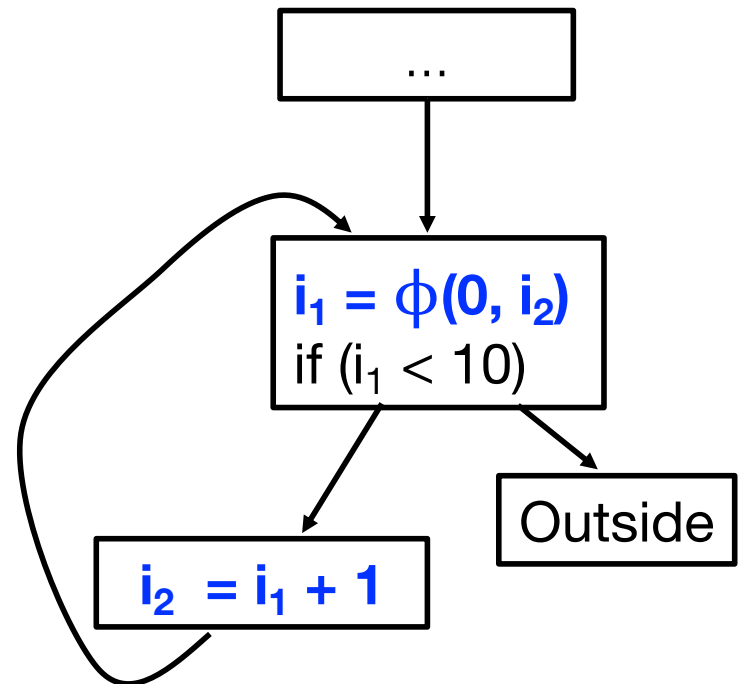
Loop Induction Variables

- ▶ An induction variable is a variable that
 - Gets increased or decreased by a fixed amount (loop invariant) on every iteration of a loop
 - $i = i + c$ (**basic induction variable**)
 - or is an affine function of another induction variable
 - $j = a * i + b$ (**mutual induction variable**)

Identifying Basic Loop Induction Variable

► Find basic loop induction variable(s)

1. Inspect back edges in the loop
2. Each back edge points to a ϕ node in the loop header, which may indicate a basic induction variable
3. ϕ is a function of an initialized variable and a definition in the form of “ $i + c$ ” (i.e., increment operation)



Next Lecture

- ▶ Scheduling

Acknowledgements

- ▶ These slides contain/adapt materials developed by
 - Prof. Scott Mahlke (UMich)