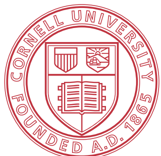
ECE 6775
High-Level Digital Design Automation
Fall 2025

Static Single Assignment



Cornell University

Announcements

- Lab 2 is out, and a relevant note on loop unrolling note is posted on Ed

Agenda

- More control flow analysis
  - Dominator tree
  - Dominance frontier

- Data flow graph (DFG) and data dependences

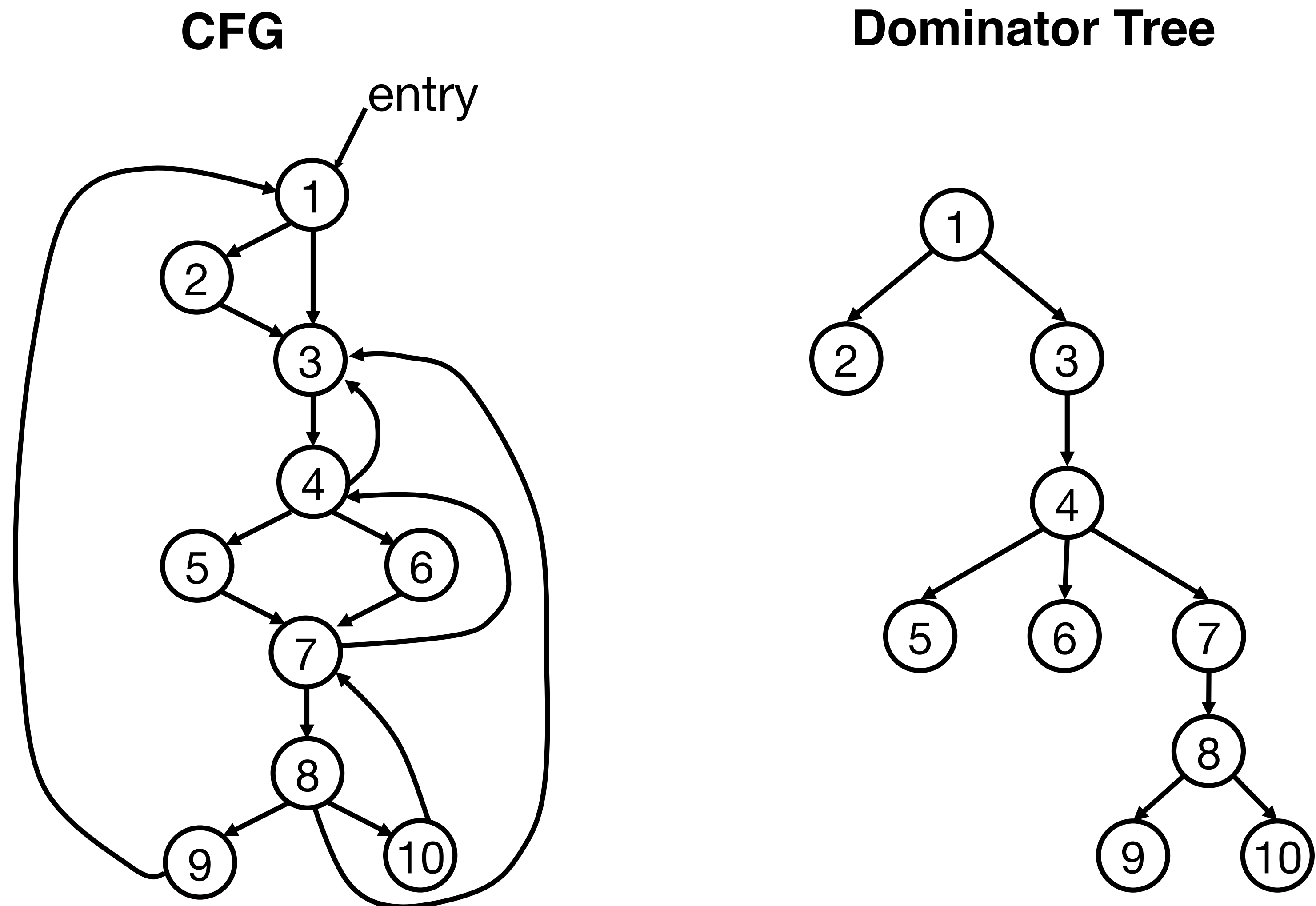
- Static single assignment (SSA)
  - SSA definition
  - PHI node (Φ-node) placement
  - Code optimizations with SSA

- A brief overview of LLVM

Control Flow Graph: True or False?

- The dominator set of a CFG node can be empty
- A loop must be a strongly connected component (SCC) in CFG
- Each CFG node can only have up to one immediate dominator

Review: Dominator Tree



Dominator trees can be efficiently constructed in near-linear time

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

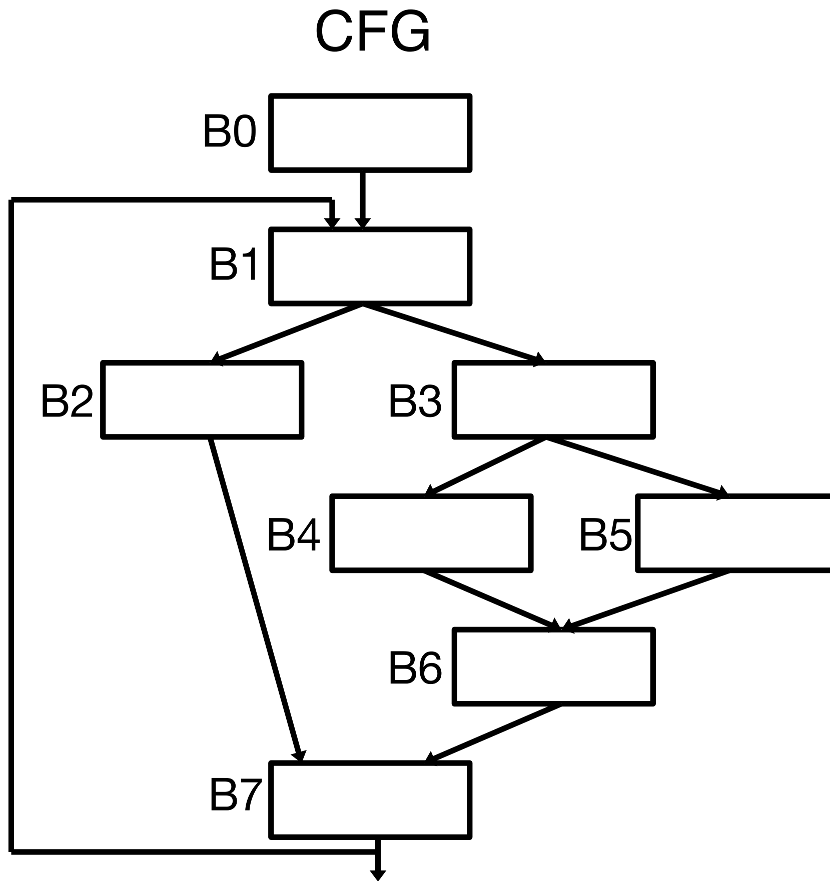
  (2) p dominates some predecessor(s) of q

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

- Informally, if $q \in \mathrm{DF}(p)$, q is *almost strictly dominated* by p

- Useful for efficiently computing the SSA form

Example: Dominance Frontiers



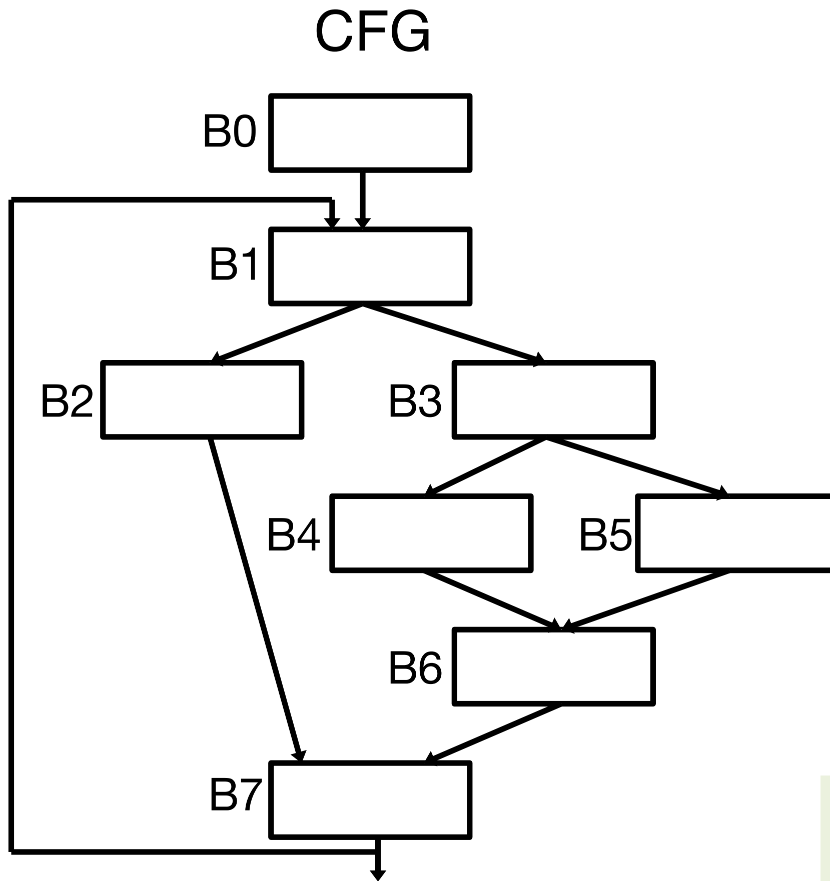
Dominance frontiers (DF)

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, $q \in \mathrm{DF}(p)$

Dominance Frontiers and Dominator Tree

CFG

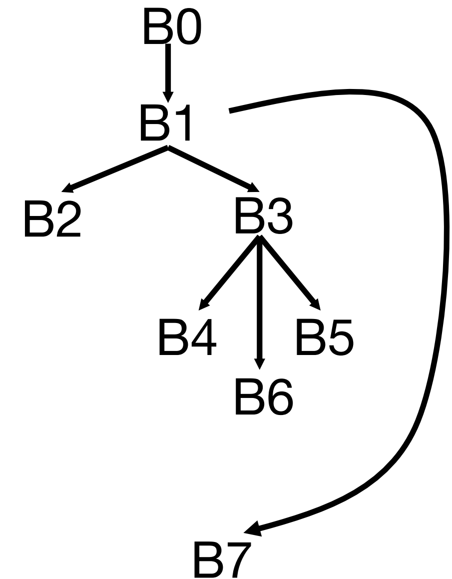


A **convergence point** is a node in CFG with multiple predecessors (B1, B6, B7 in this example)

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 = 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

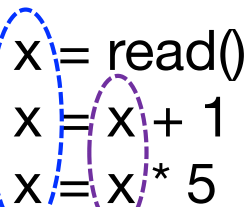
- The **static single assignment (SSA) form** is a restricted IR where
  - Each (scalar) variable definition has a **unique** name
  - Each (scalar) variable use refers to a single definition

- SSA simplifies both dataflow and dependence analyses, enabling more effective and streamlined compiler optimizations
  - SSA eliminates artificial dependences (e.g., WAW, WAR) on on scalars

SSA within a Basic Block

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

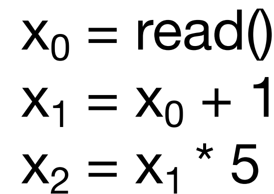
Original code



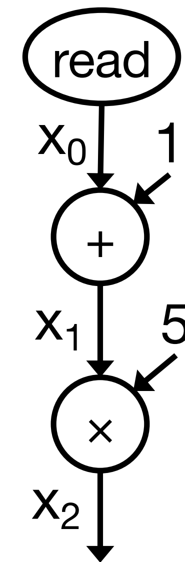
Defs of x

Uses of x

SSA form

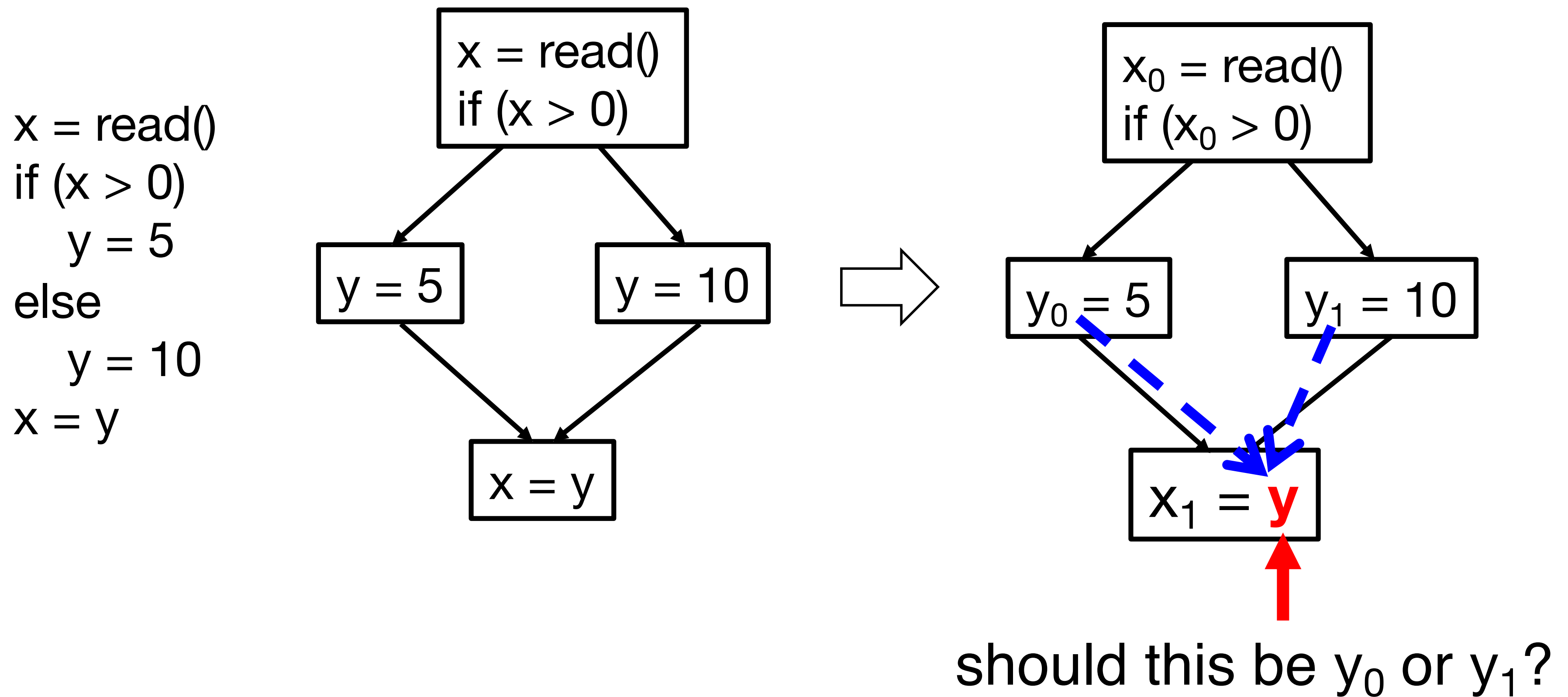


Corresponding data flow graph



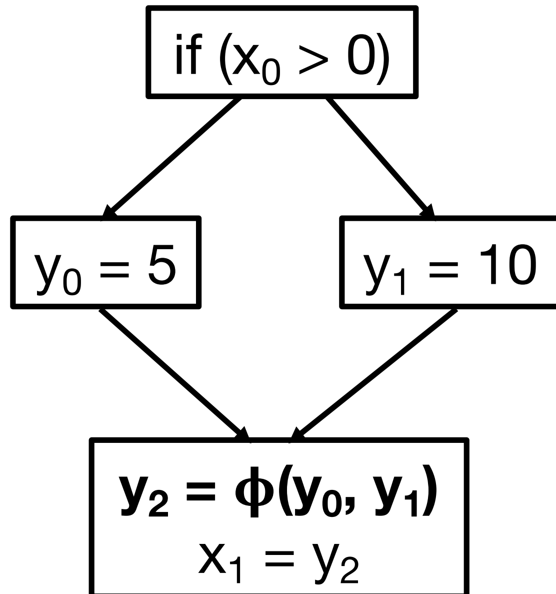
SSA with Control Flow

- 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



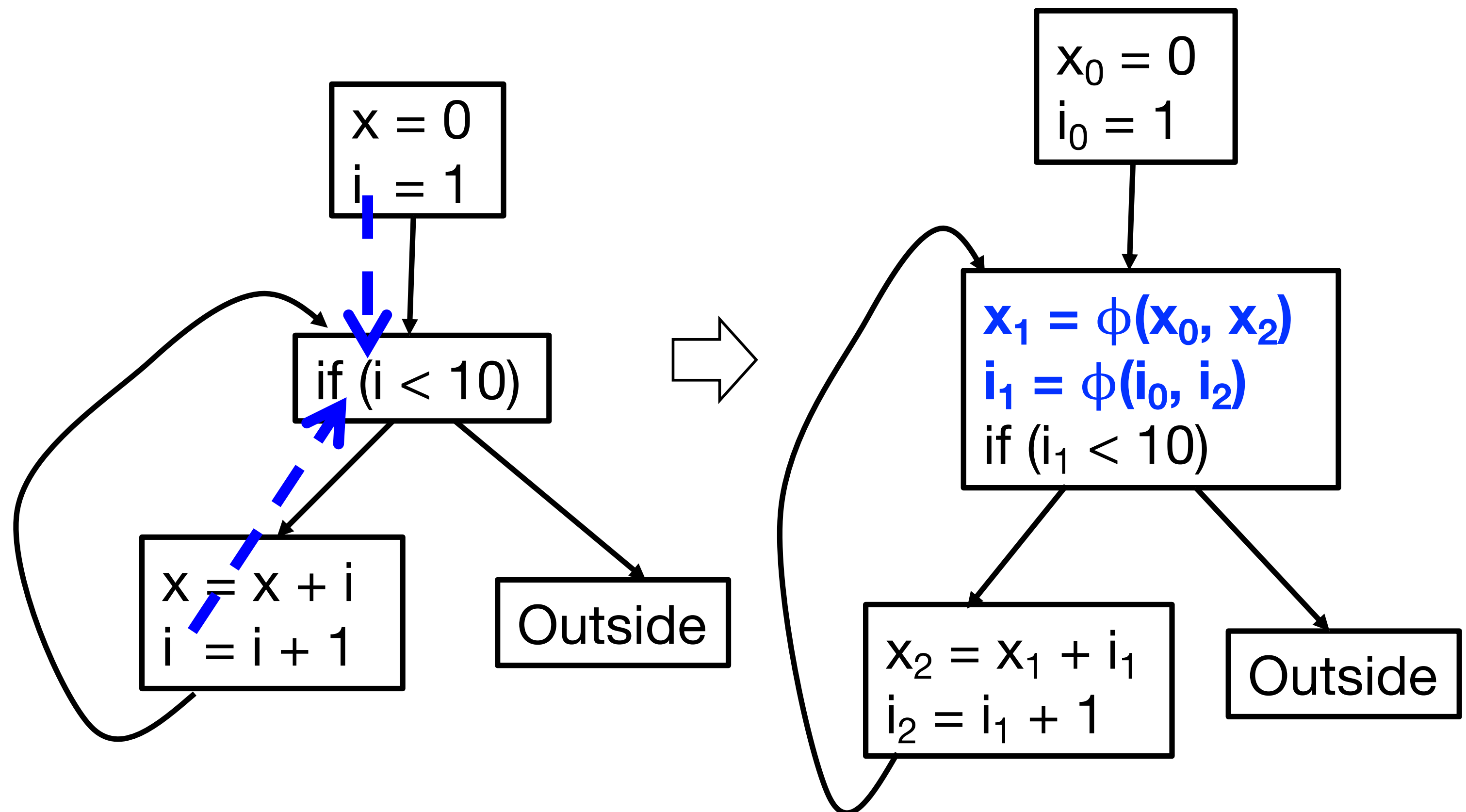
Note: ϕ is not an executable function!

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

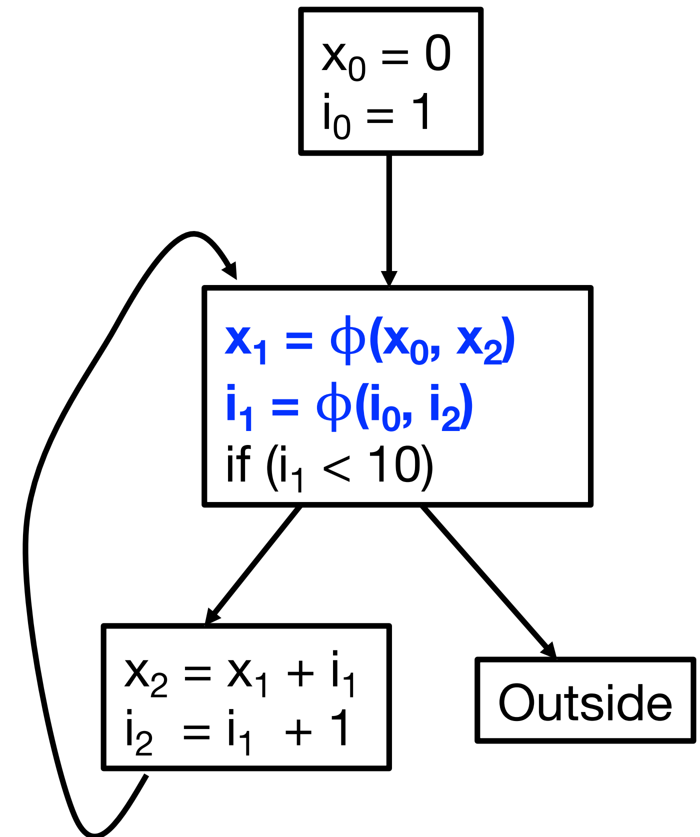
- Insert ϕ-nodes in the loop header block

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

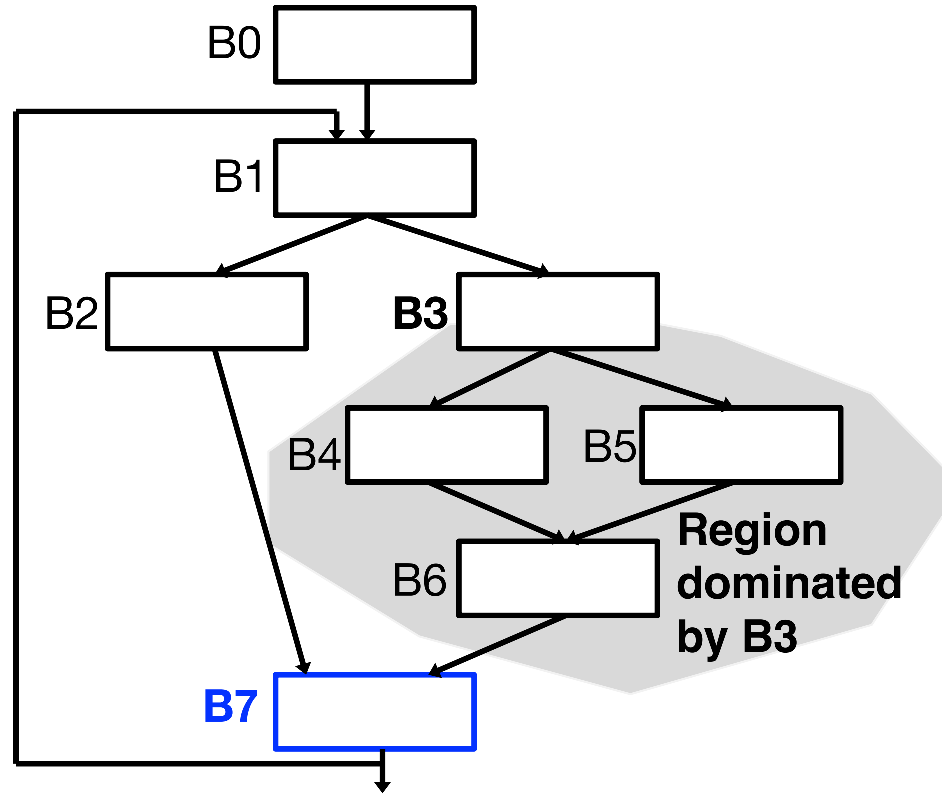


ϕ–Node Placement

- When and where to insert ϕ-nodes?
  - If two control paths A → C and B → 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



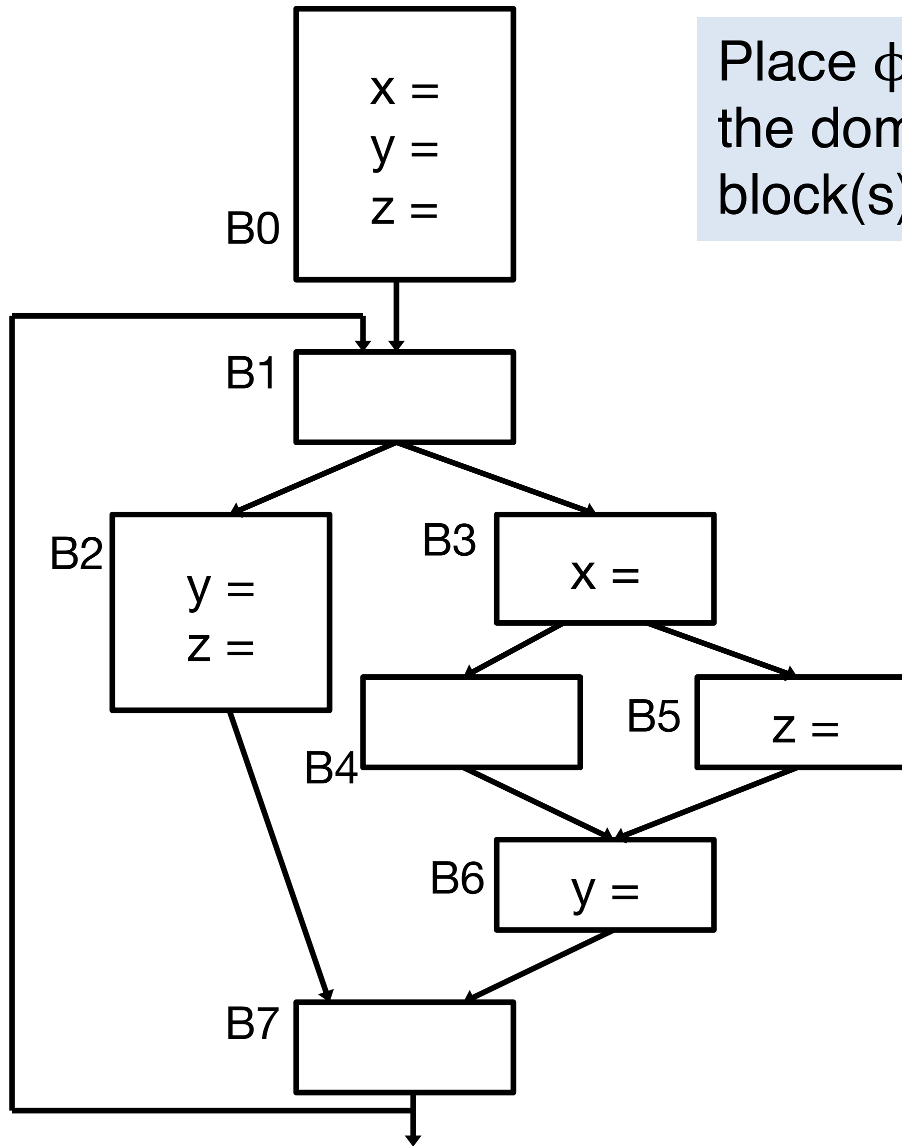
Example: Dominance Frontier and SSA



- 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

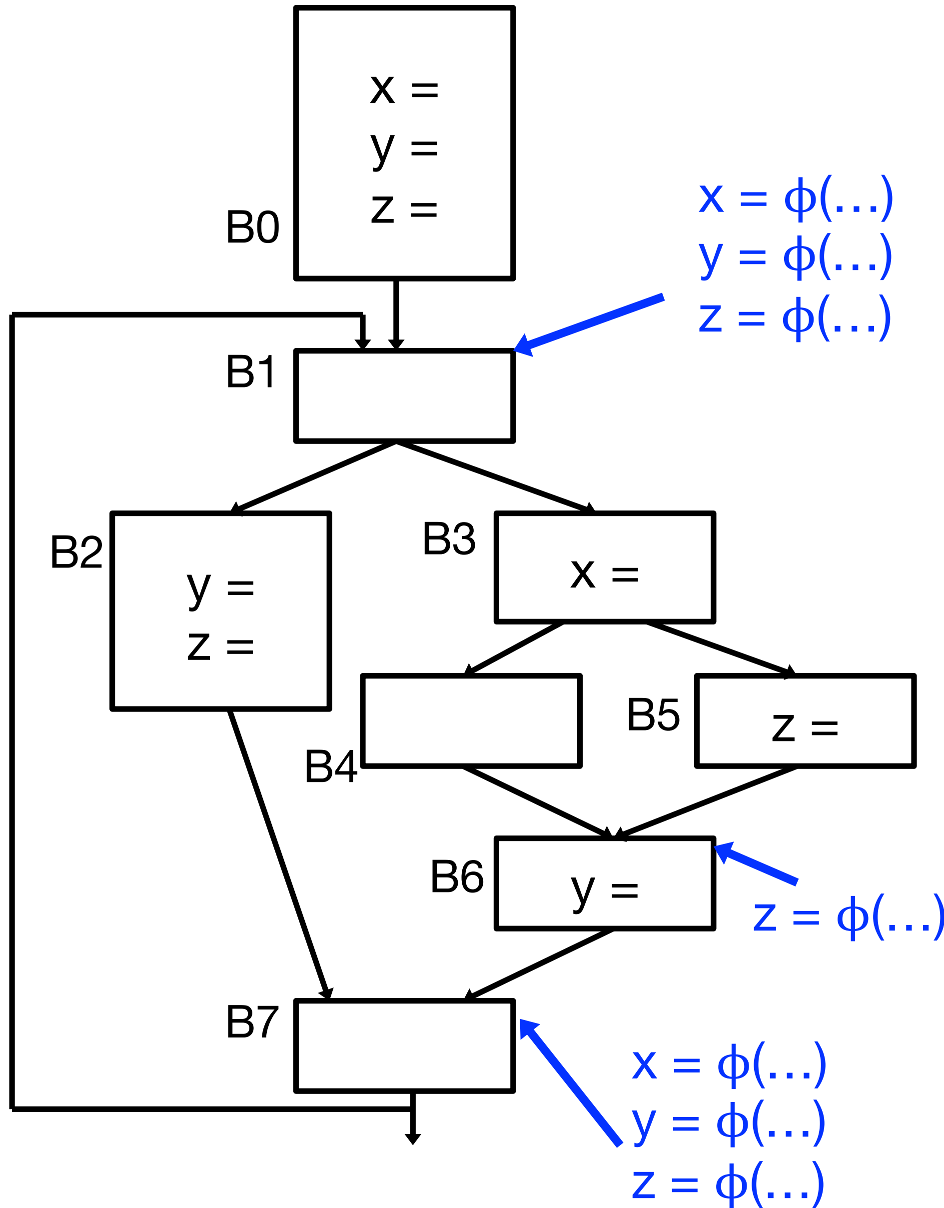
p	**DF**
0	-
1	1
2	7
3	7
4	6
5	6
6	7
7	1



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

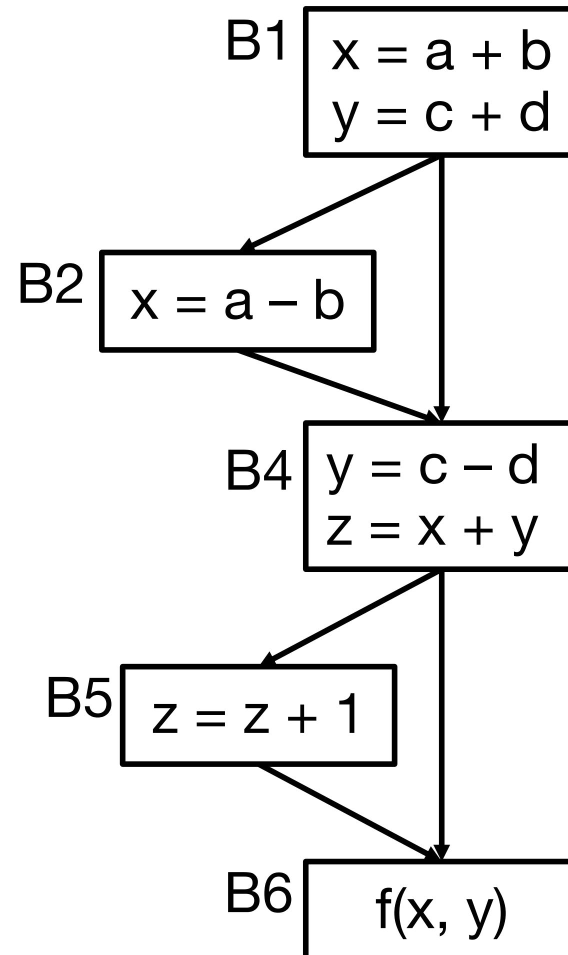
SSA Applications

- SSA form simplifies data flow analysis and many code transformations
  - Primarily due to explicit & simplified (sparse) def-use chains

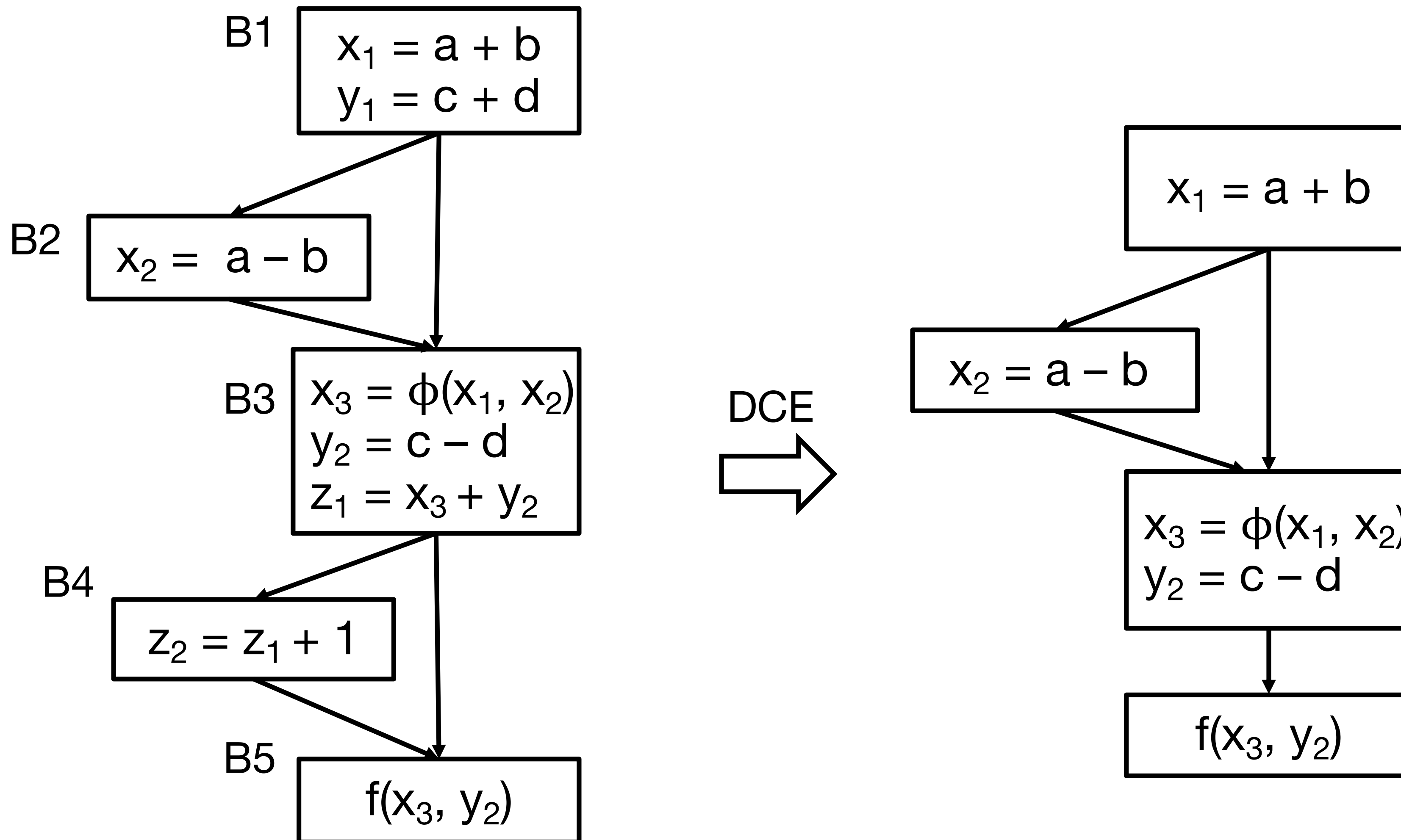
- Here we show two simple examples
  - Dead code elimination
  - Loop induction variable detection

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 y1, z2, and B4; then remove z1

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**)

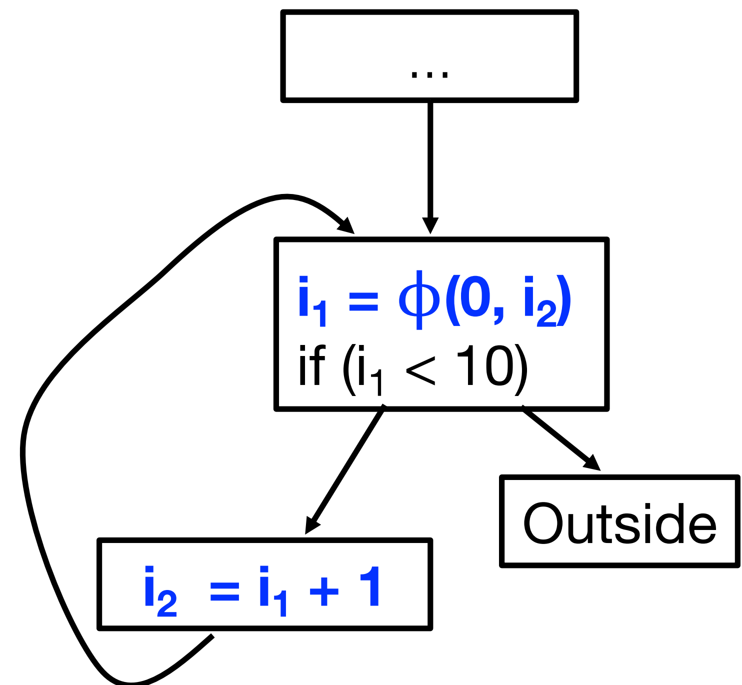
Identifying Basic Loop Induction Variable

- 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)



LLVM: A Foundational IR for Modern Compilers

- Formerly Low Level Virtual Machine
  - Brainchild of Chris Lattner and Vikram Adve back in 2000
  - ACM Software System Award in 2012
  - LLVM is widely used in industry and academia (including the HLS tool in our class)

- **The core of LLVM is the SSA-based IR**
  - Language independent, target independent, easy to use
  - RISC-like virtual instructions, unlimited registers, exception handling, etc.

- Provides modular & reusable components for building compilers
  - Components are largely language- and target-independent
  - Rich libraries with clean interfaces (optimizations, analyses, codegen, profiling, etc.)
  - Tools built on libraries: Clang (C/C++/ObjC compiler), Apple Swift, Rust, Vivado HLS, and many others

Example: An LLVM Loop

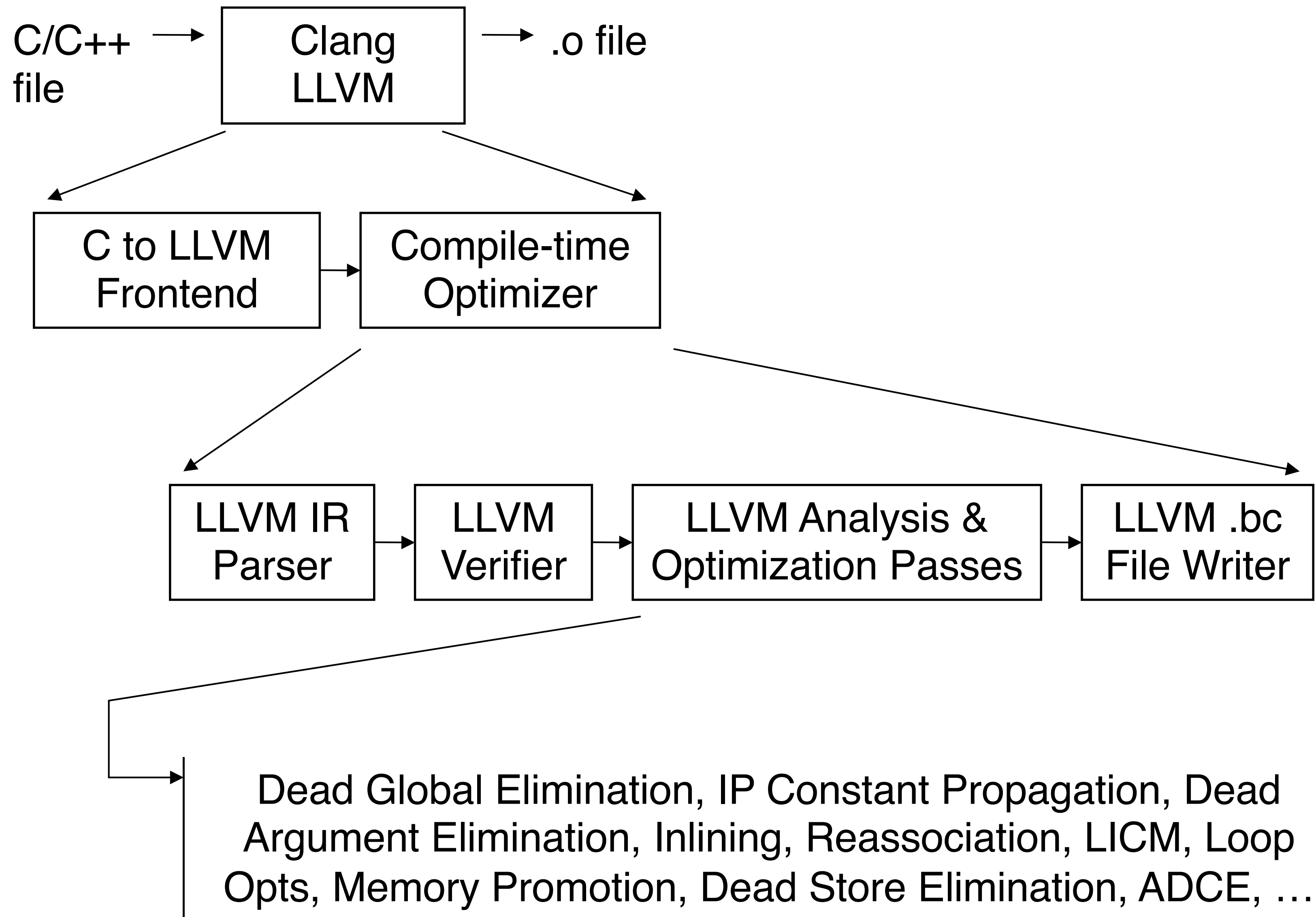
```
for (i=0; i<N; ++i)
   foo(A[i], &P);
```

```
loop:
  %i.1 = phi i5 [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, i32 %i.1
  call void %foo(float %AiAddr, %pair* %P)
  %i.2 = add i5 %i.1, 1
  %tmp = icmp eq i5 %i.1, 16
  br i1 %tmp, label %loop, label %outloop
```

- High-level information exposed in the code
  - Explicit dataflow through SSA form
  - Explicit control-flow graph
  - Explicit language-independent type-information
  - Explicit typed pointer arithmetic
    - Preserve array subscript and structure indexing

source: http://llvm.org

An LLVM-Based Compiler



source: http://llvm.org

Next Lecture

- Scheduling

Acknowledgements

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