

# Compiler Optimization

# Compiler optimizations transform code

- ❑ Code optimization transforms code to equivalent code
  - ... but with better performance
- ❑ The code transformation can involve either
  - **Replacing** code with more efficient code
  - **Deleting** redundant code
  - **Moving** code to a position where it is more efficient
  - **Inserting** new code to improve performance

# The four categories of code transformations

- ❑ Replacing code (e.g. **strength reduction**)

$A=2*a;$      $\equiv$      $A=a\ll 1;$

- ❑ Deleting code (e.g. **dead code elimination**)

$A=2; A=y;$      $\equiv$      $A=y;$

- ❑ Moving code (e.g. **loop invariant code motion**)

`for (i = 0; i < 100; i++) { sum += i + x * y; }`

$\equiv$

$t = x * y;$

`for (i = 0; i < 100; i++) { sum += i + t; }`

- ❑ Inserting code (e.g. **data prefetching**)

`for (p = head; p != NULL; p = p->next)`

`{ /* do work on node p */ }`

$\equiv$

`for (p = head; p != NULL; p = p->next)`

`{ prefetch(p->next); /* do work on node p */ }`

# Compiler optimization categories according to range

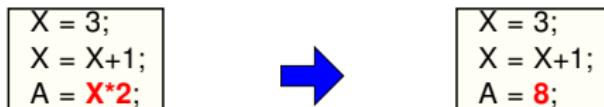
- ❑ How much code does the compiler view while optimizing?
  - The wider the view, the more powerful the optimization
- ❑ Axis 1: optimize across control flow?
  - **Local optimization**: optimizes only within straight line code
  - **Global optimization**: optimizes across control flow  
(if,for,...)
- ❑ Axis 2: optimize across function calls?
  - **Intra-procedural optimization**: only within function
  - **Inter-procedural optimization**: across function calls
- ❑ The two axes are orthogonal (any combination is possible)

# Local vs. Global Constant Propagation

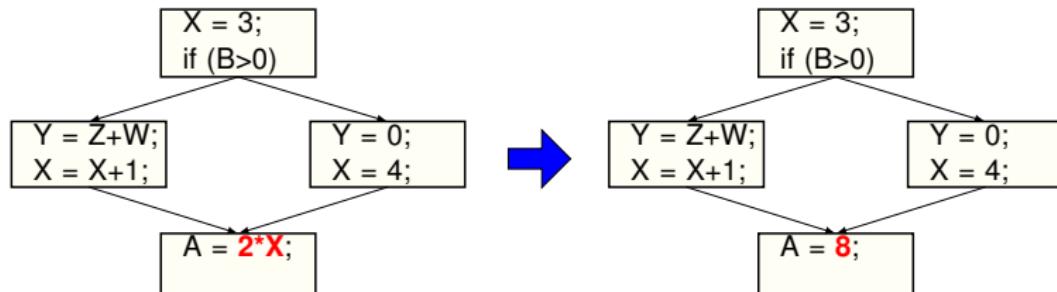
## ❑ Constant propagation

- Optimization: if  $x = y \text{ op } z$  and y and z are constants then compute at compile time and replace

## ❑ Local Constant Propagation



## ❑ Global Constant Propagation



- Additional **control flow analysis** is needed to enable this.

# Intra- vs. Inter-procedural Constant Propagation

## Intra-procedural Constant Propagation

```
X = 3;  
X = X+1;  
A = X*2;
```



```
X = 3;  
X = X+1;  
A = 8;
```

- Above is a local intra-procedural constant propagation.

## Inter-procedural Constant Propagation

```
X = 3;  
foo();  
A = X*2;
```

```
void foo() {  
    // No update to X.  
}
```



```
X = 3;  
foo();  
A = 6;
```

```
void foo() {  
    // No update to X.  
}
```

- Above is a local inter-procedural constant propagation.
- The fact that global variable X is not updated must be propagated from callee to caller function to enable this.

# Control Flow Analysis

# Basic Block

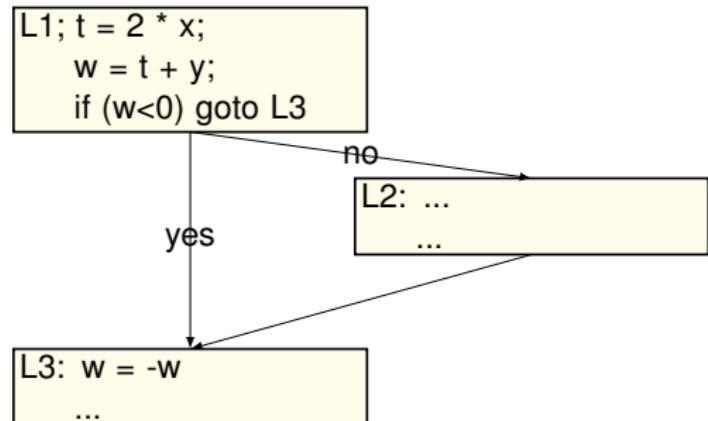
- ❑ A function body is composed of one or more **basic blocks**.
- ❑ **Basic block:** a maximal sequence of instructions that either execute all together or none at all. Meaning:
  - There are no jumps into the middle of the basic block
  - There are no jumps out of the middle of the block
- ❑ That means:
  - No instruction other than the first is a jump target
  - No instruction other than the last is a jump or branch
- ❑ Either all instructions in basic block execute or none
  - Smallest unit of execution in control flow analysis
  - Hence the descriptor "basic" in the name

# Control Flow Graph

- ❑ A **Control Flow Graph (CFG)** is a directed graph in which
  - Nodes are basic blocks
  - Edges represent flows of execution between basic blocks
- ❑ CFGs are widely used to represent a program for analysis
- ❑ CFGs are especially essential for global optimizations

# Control Flow Graph Example

```
L1; t = 2 * x;  
    w = t + y;  
    if (w<0) goto L3  
L2: ...  
...  
L3: w = -w  
...
```



# Construction of CFG

- ❑ Step 1: partition code into basic blocks
  - Identify **leader** instructions, where a leader is either:
    - the first instruction of a program, or
    - the target of any jump/branch, or
    - an instruction immediately following a jump/branch
  - Create a basic block out of each leader instruction
  - Expand basic block by adding subsequent instructions  
(Stopping when the next leader instruction is encountered)
  
- ❑ Step 2: add edge between two basic blocks B1 and B2 if
  - there exist a jump/branch from B1 to B2, or
  - B2 follows B1, and B1 does not end with unconditional jump

# Example

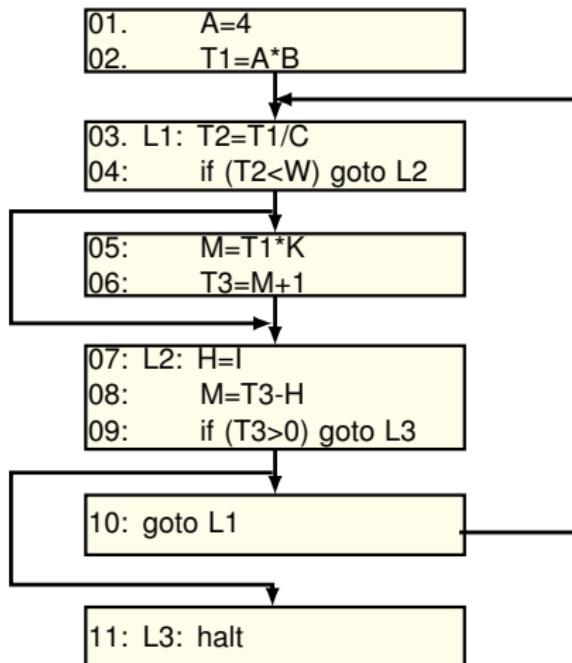
```
01.    A=4
02.    T1=A*B
03. L1: T2=T1/C
04:    if (T2<W) goto L2
05:    M=T1*K
06:    T3=M+1
07: L2: H=I
08:    M=T3-H
09:    if (T3>0) goto L3
10:   goto L1
11: L3: halt
```

# Example

```
01.    A=4
02.    T1=A*B
03. L1: T2=T1/C
04.    if (T2<W) goto L2
05.    M=T1*K
06.    T3=M+1
07. L2: H=I
08.    M=T3-H
09.    if (T3>0) goto L3
10.   goto L1
11. L3: halt
```

# Example

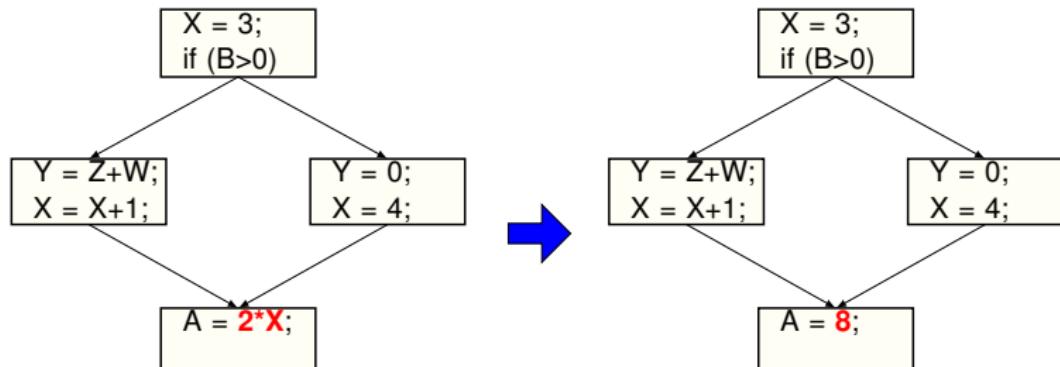
```
01. A=4
02. T1=A*B
03. L1: T2=T1/C
04: if (T2<W) goto L2
05: M=T1*K
06: T3=M+1
07: L2: H=I
08: M=T3-H
09: if (T3>0) goto L3
10: goto L1
11: L3: halt
```



# Data Flow Analysis

# Global Optimizations

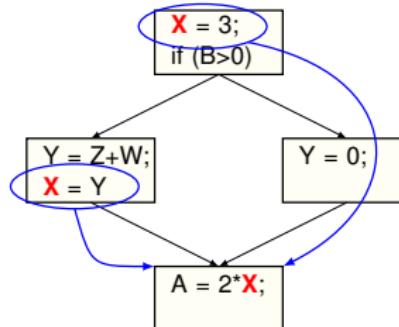
- ❑ Extends optimizations across control flows, i.e. CFG
- ❑ Like in this example Global Constant Propagation (GCP):



- ❑ How do we know it is OK to globally propagate constants?

# Correctness criteria for GCP

- ❑ There are situations that prohibit GCP:



- ❑ To replace  $X$  by a constant  $C$  **correctly**, we must know
  - **Along all paths**, the last assignment to  $X$  is " $X = C$ "
- ❑ Paths may go through loops and/or branches
  - When two paths **meet**, need to make **conservative** choice

# Global Optimizations need to be Conservative

- ❑ Many compiler optimizations depend on knowing some property X at a particular point in program execution
  - Need to prove at that point property X holds along all paths
- ❑ To ensure correctness, optimization must be **conservative**
  - An optimization is enabled only when X is definitely true
  - If not sure, be conservative and say **don't know**
  - **Don't know** typically disables the optimization

# Dataflow Analysis Framework

❑ **Dataflow analysis:** discovering properties about values at each statement of the program

- E.g. discovering a value is constant before a statement
- Done by observing the flow of data through the CFG

❑ **Dataflow analysis framework:**

- A framework for implementing various dataflow analyses
- 4 parameters defining analysis is passed into framework:

$$\{ \mathbf{D}, \mathbf{V}, \wedge: (\mathbf{V}, \mathbf{V}) \rightarrow \mathbf{V}, \mathbf{F}: \mathbf{V} \rightarrow \mathbf{V} \}$$

- **D:** direction of dataflow (forward or backward)
- **V:** domain of values denoting property
- **$\wedge$ :** **meet operator** that merges values when paths meet
- **F:** **flow propagation function** that propagates values through a basic block

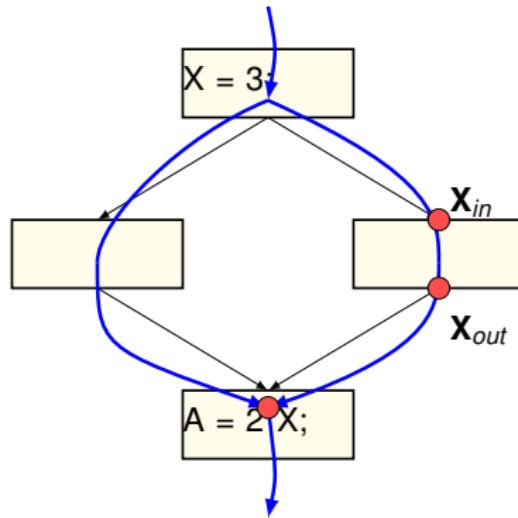
# Global Constant Propagation

# Global Constant Propagation (GCP)

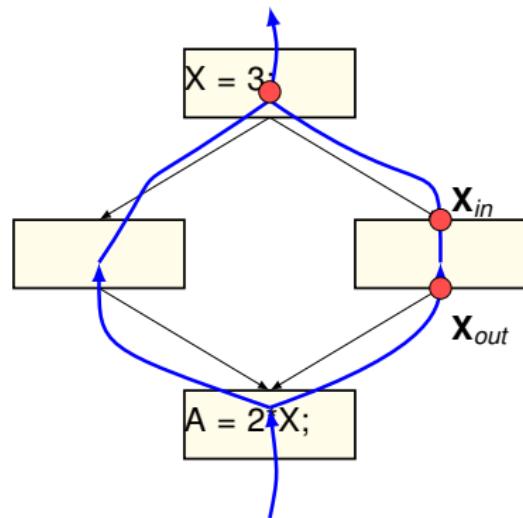
- ❑ Let's use **GCP** to study dataflow analysis framework
- ❑ We will define each component one by one for GCP
  - **D**: direction of dataflow for constant property
  - **V**: domain of values denoting constant property
  - **$\wedge$** : **meet operator** that merges values when paths meet
  - **F**: **flow propagation function** for GCP

# Direction D for GCP

- ❑ Is GCP a forward or backward analysis?



**Forward Analysis**



**Backward Analysis**

- ❑ Forward, since "constantness" of a variable flows forward to subsequent instructions starting from assignment

# Dataflow property V for GCP

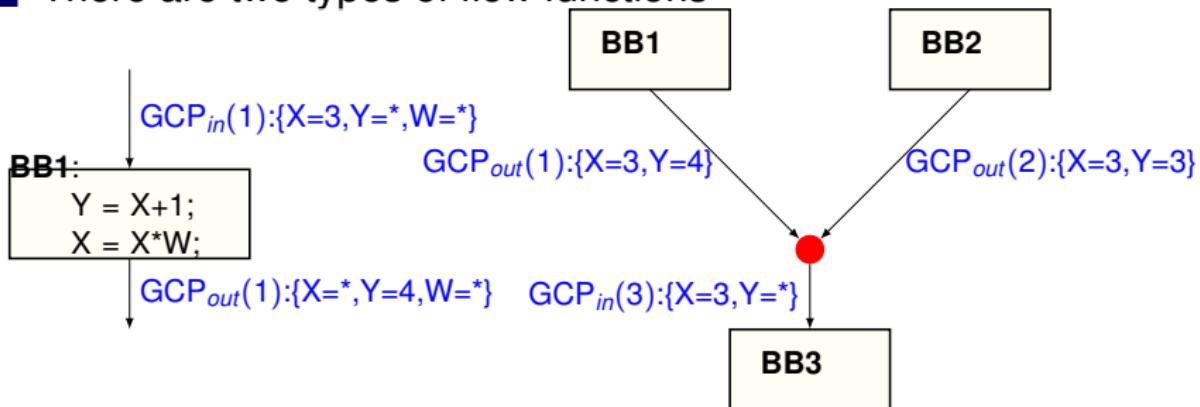
- ❑ V is a map of variables to values, where a value is:  
(in the case where value is an int type)

```
# /* not defined yet */  
..., -1, 0, 1, ... /* a constant */  
* /* not a constant */
```

- ❑ **GCP(*i*)**: GCP dataflow property of basic block *i*
  - **GCP<sub>in</sub>(*i*)**: at the entry of basic block *i*
  - **GCP<sub>out</sub>(*i*)**: at the exit of basic block *i*
- ❑ **GCP(*i*)[X]**: value mapped to variable X in GCP(*i*)
- ❑ Example: given  $\text{GCP}_{\text{in}}(1) = \{\text{X}=1, \text{Y}=\#, \text{Z}=\ast\}$ 
  - $\text{GCP}_{\text{in}}(1)[\text{X}] = 1, \text{GCP}_{\text{in}}(1)[\text{Y}] = \#, \text{GCP}_{\text{in}}(1)[\text{Z}] = \ast$

# Dataflow Equations for GCP

- ❑ There are two types of flow functions



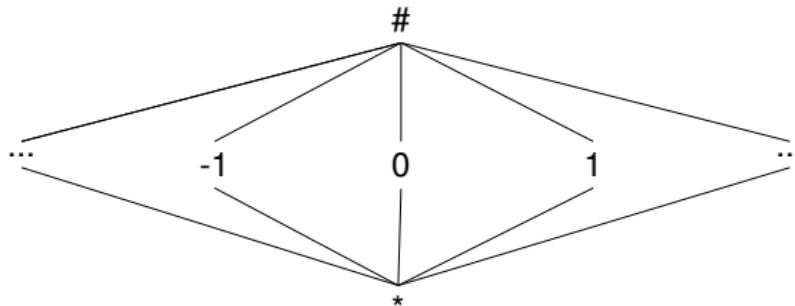
- Flow transfer function  $F: V \rightarrow V$ 
  - Computes data flow across statements
  - If statement assigns  $X$ , update  $GCP_{out}(i)[X]$  accordingly
- Meet operator  $\wedge: (V, V) \rightarrow V$ 
  - Computes data flow at control flow merges
  - Merge property from two paths using the meet operator

# Flow Transfer Function F for GCP

- ❑ Treat each statement as basic block  $i$  to apply  $F$
- ❑ If statement is not an assignment,  $GCP_{out}(i) = GCP_{in}(i)$
- ❑ If statement is of the form  $X = Y + Z$ ,
  - If  $GCP_{in}(i)[Y]$  and  $GCP_{in}(i)[Z]$  are both constants,  
 $GCP_{out}(i)[X] = GCP_{in}(i)[Y] + GCP_{in}(i)[Z]$
  - Else if either  $GCP_{in}(i)[Y]$  or  $GCP_{in}(i)[Z]$  is  $*$ ,  
 $GCP_{out}(i)[X] = *$
  - Else if either  $GCP_{in}(i)[Y]$  or  $GCP_{in}(i)[Z]$  is  $\#$ ,  
 $GCP_{out}(i)[X] = \#$

# Meet operator $\wedge$ for GCP

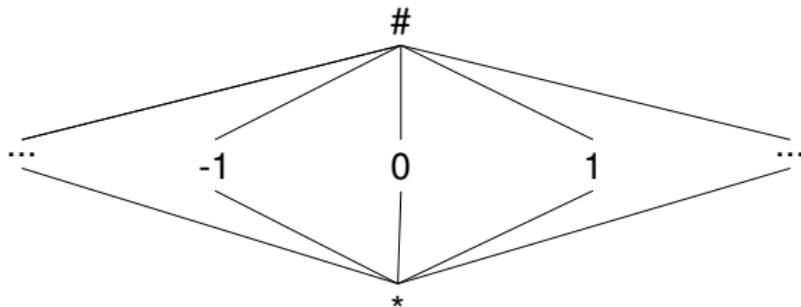
- ❑ Given basic block 1 and 2 merge into basic block 3,  
 $GCP_{in}(3) = GCP_{out}(1) \wedge GCP_{out}(2)$ 
  - Where  $\wedge$  is applied to each variable X in  $GCP_{in}(3)$ :  
 $GCP_{in}(3) = GCP_{out}(1)[X] \wedge GCP_{out}(2)[X]$
- ❑ Meet operator  $\wedge$  is given by this **semi-lattice**:
  - $a \wedge b = \text{greatest lower bound (glb)}$  in the below graph



- # is called the **top** value denoted as  $\top$
- \* is called the **bottom** value denoted as  $\perp$

# Meet operator $\wedge$ for GCP

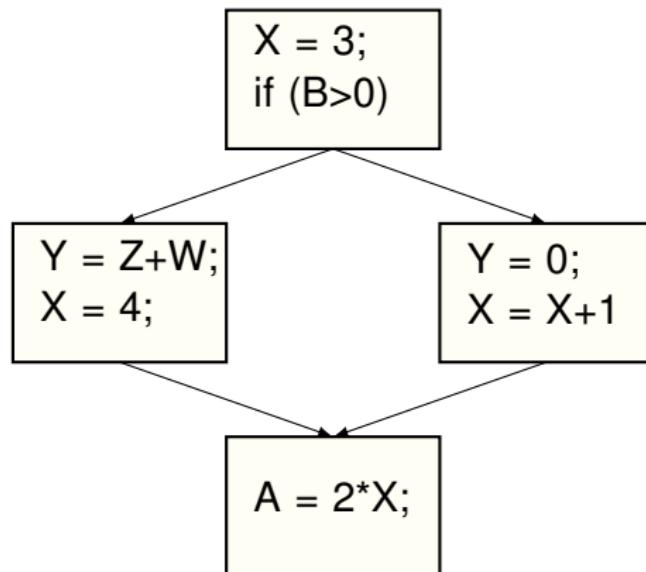
- ❑ Some results of meets  $\wedge$  given by this **semi-lattice**:



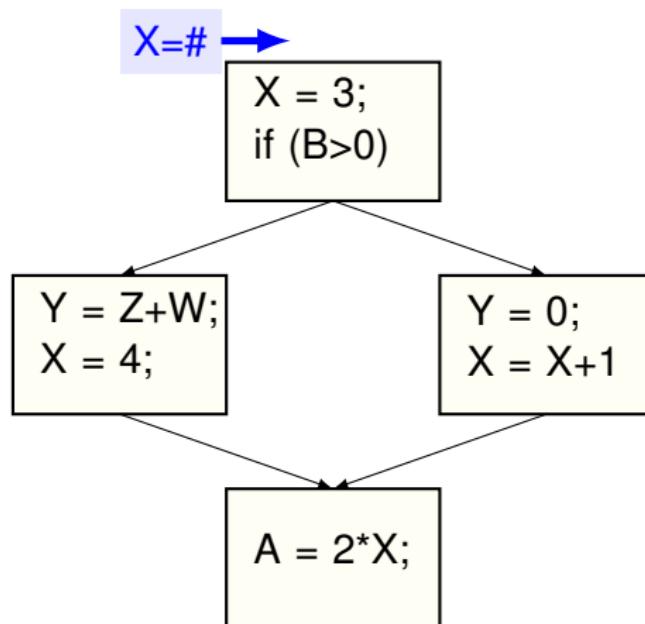
- $\# \wedge 1 \equiv \text{glb}(\#, 1) \equiv 1$ 
  - Meet of undefined value and a constant  $\rightarrow x$  is that constant
- $0 \wedge 1 \equiv \text{glb}(0, 1) \equiv *$ 
  - Meet on different constants  $\rightarrow x$  is no longer constant
- $* \wedge 1 \equiv \text{glb}(*, 1) \equiv *$ 
  - Meet of not a constant and a constant  $\rightarrow x$  is not constant

- ❑ Greatest lower bound finds the maximal conservative value

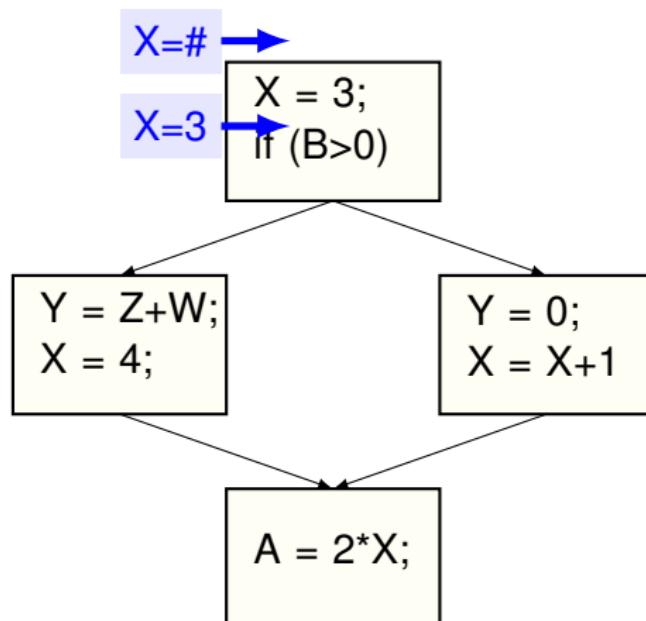
# GCP Propagation without loops



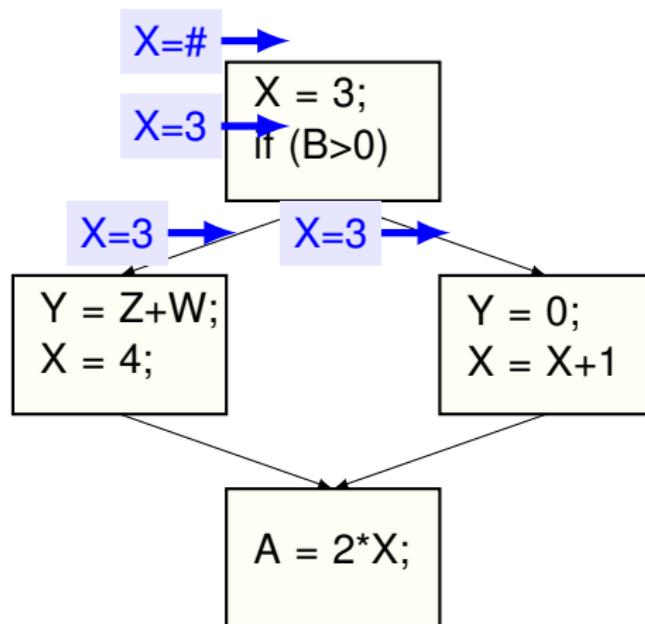
# GCP Propagation without loops



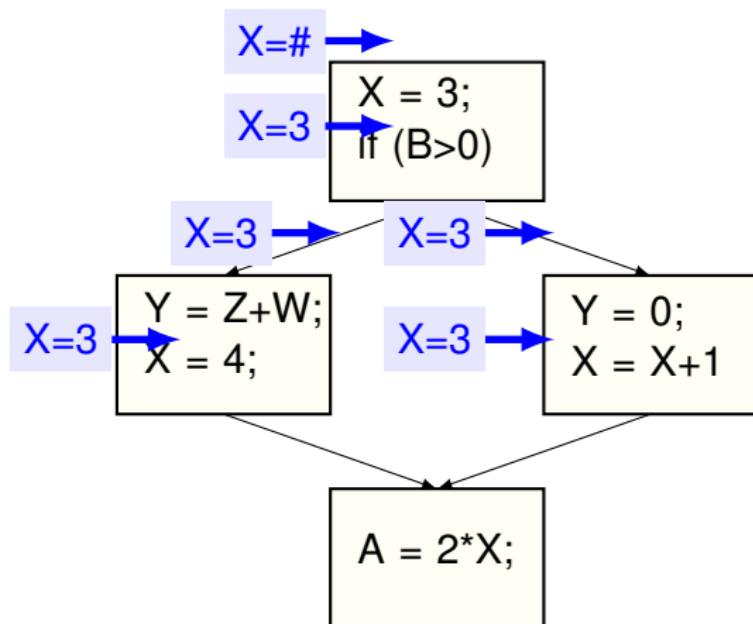
# GCP Propagation without loops



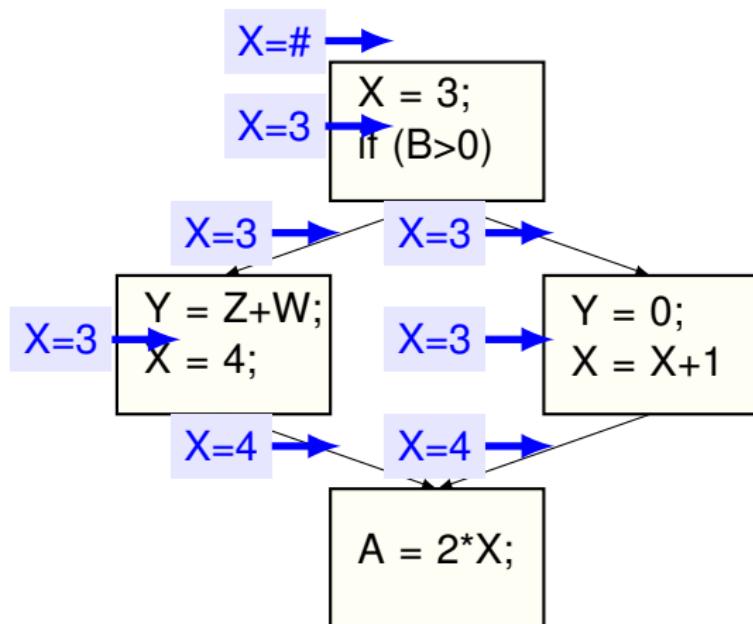
# GCP Propagation without loops



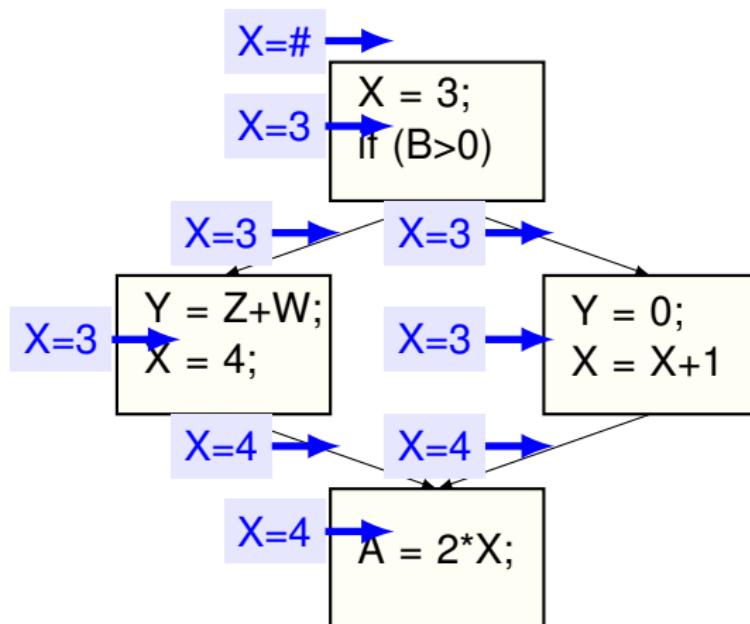
# GCP Propagation without loops



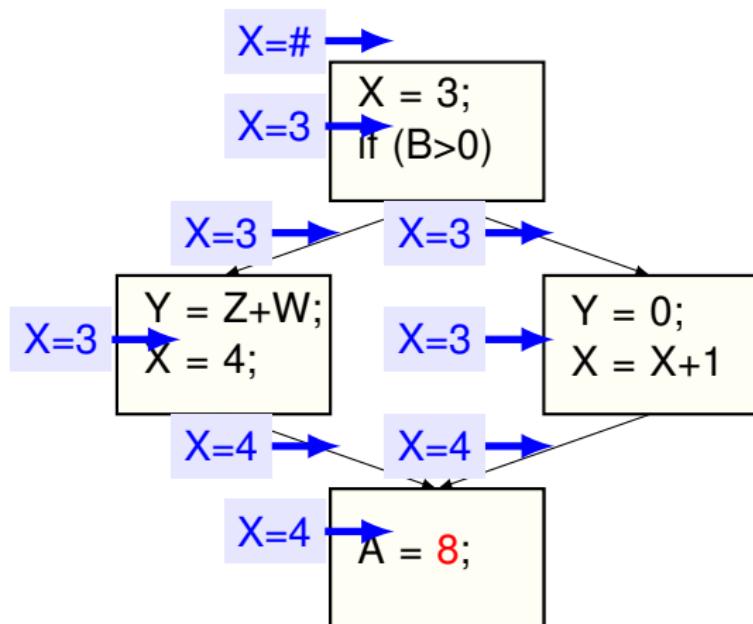
# GCP Propagation without loops



# GCP Propagation without loops

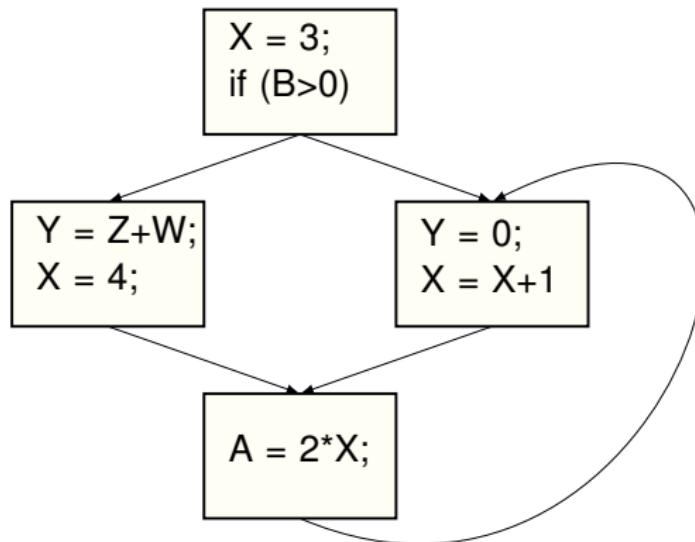


# GCP Propagation without loops



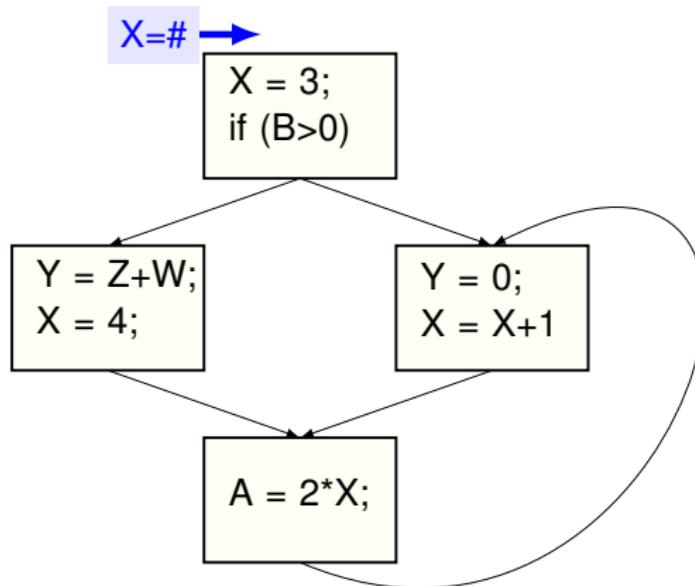
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



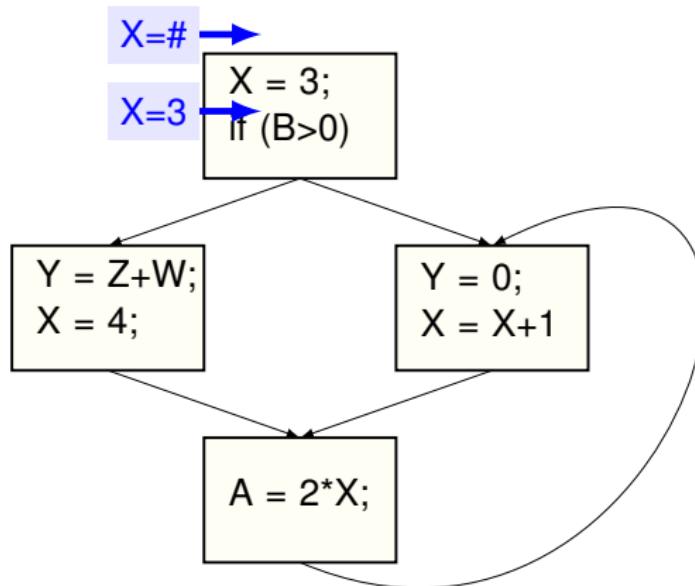
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



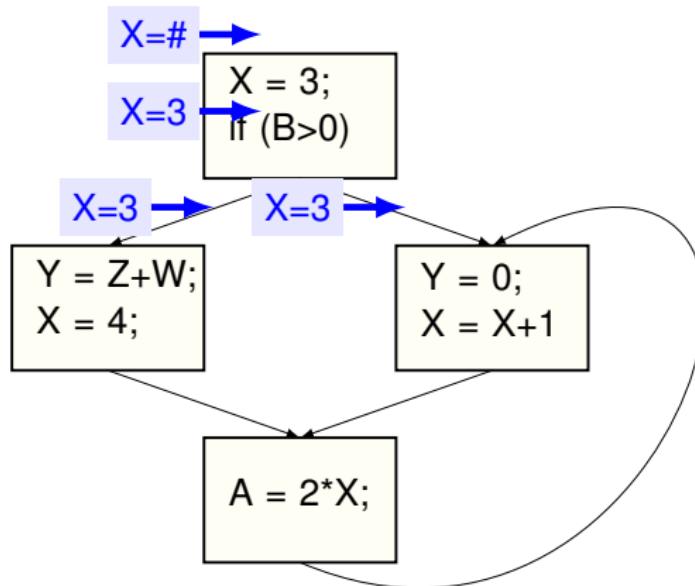
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



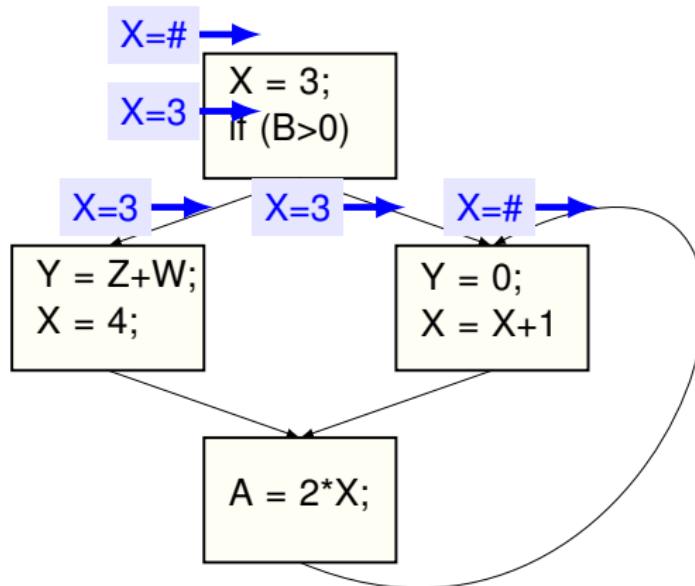
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



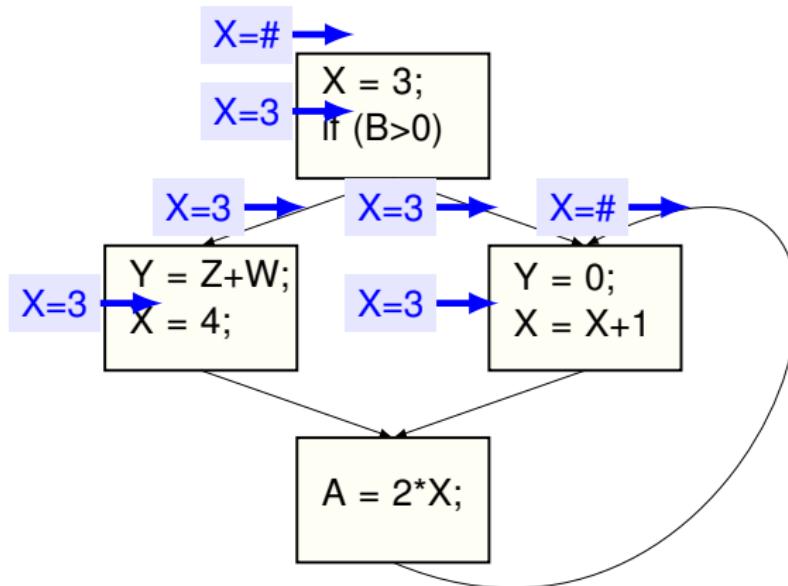
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



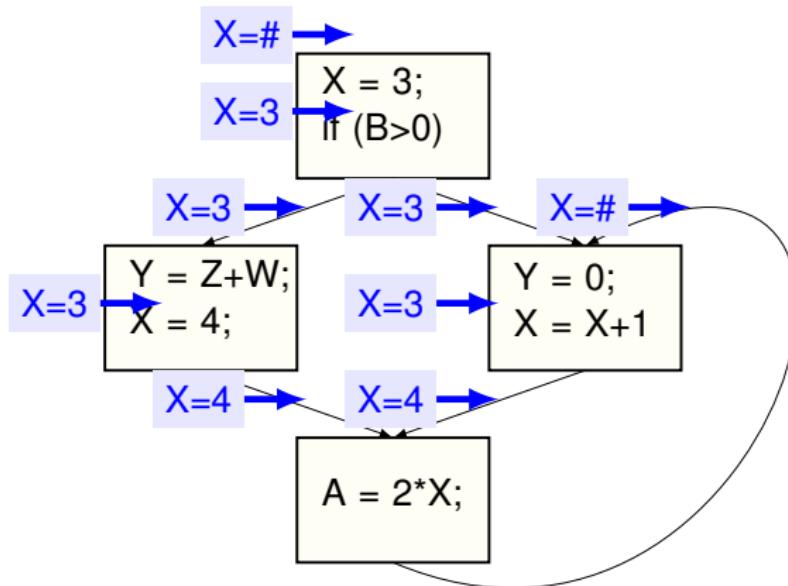
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



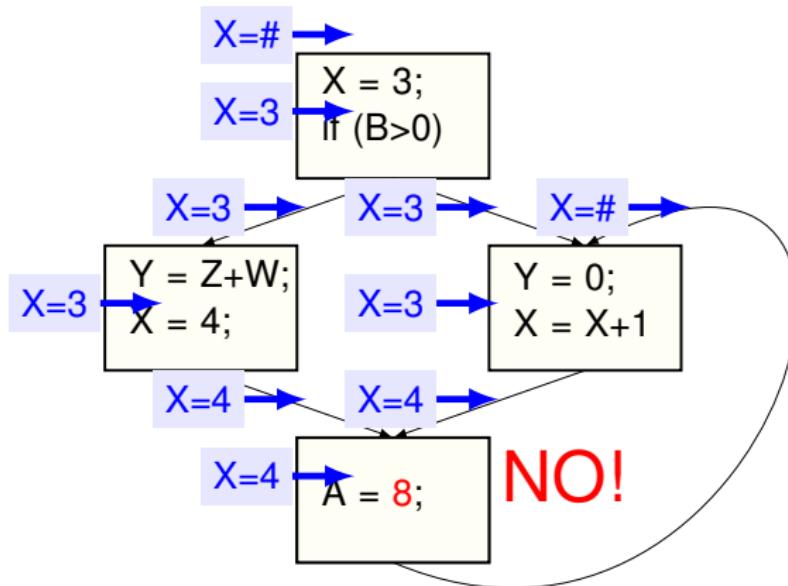
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



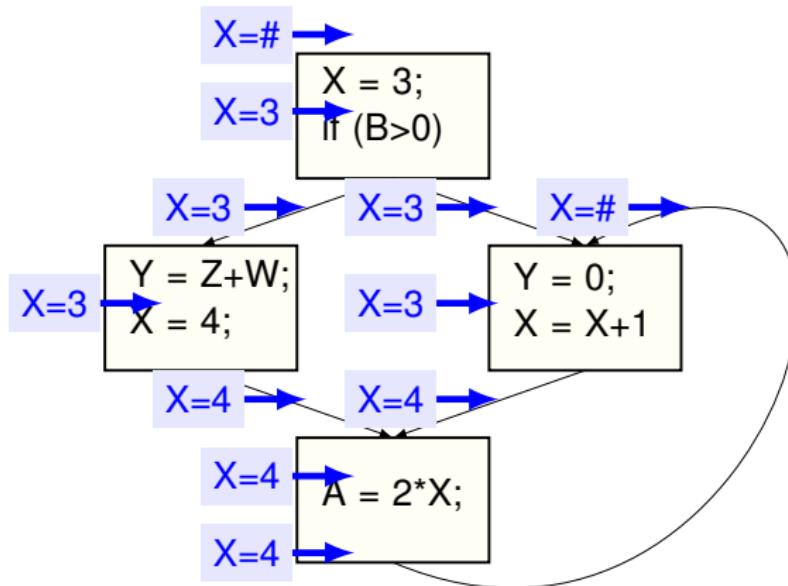
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



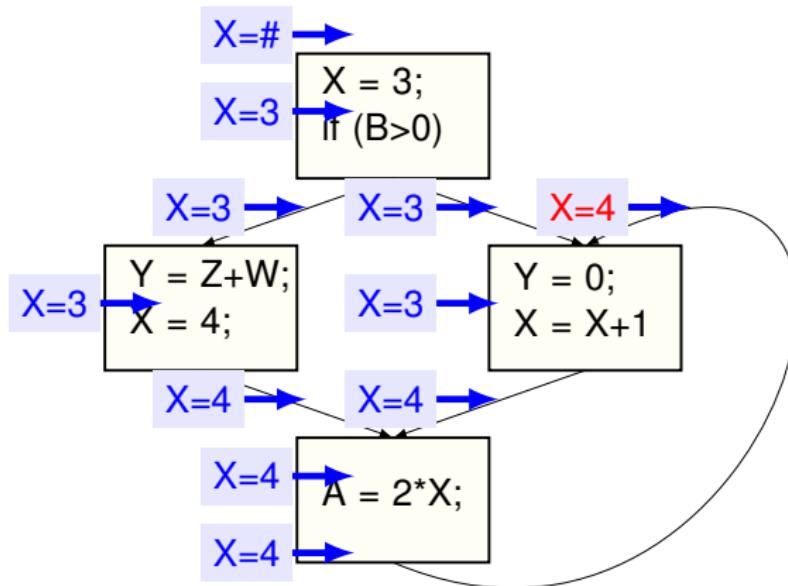
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



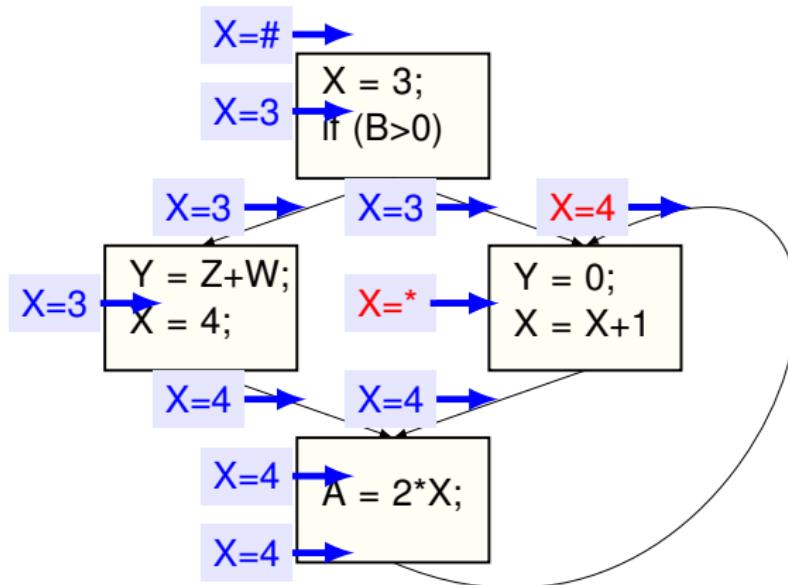
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



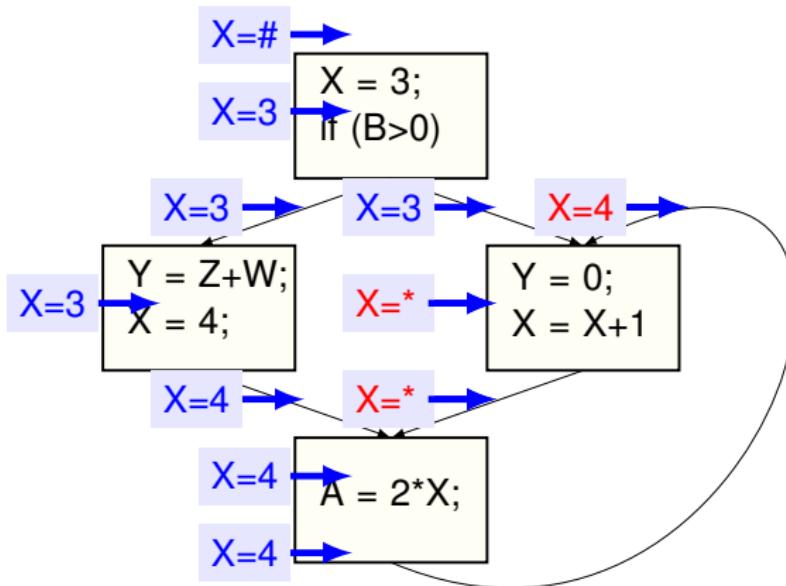
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



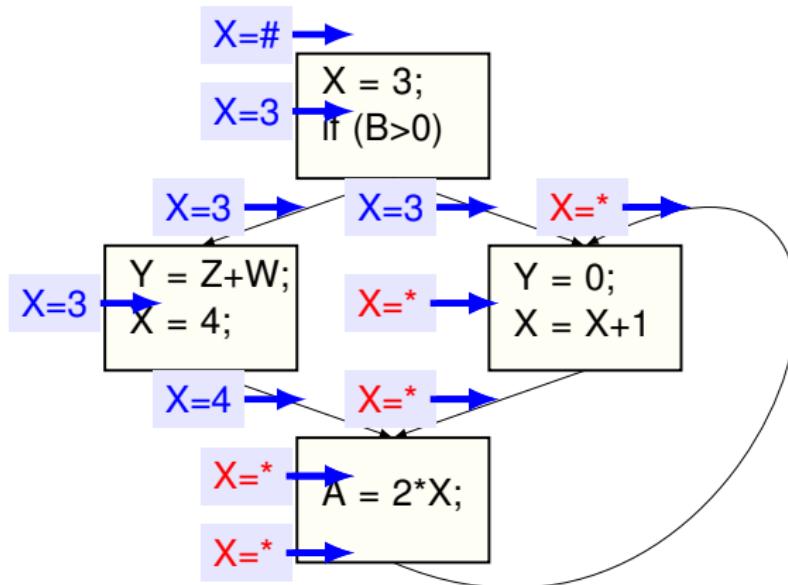
# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



# GCP Propagation with loops

- ❑ Iterate until there are no changes to values
  - This is called the **maximum fixed point** solution



# Worklist Algorithm for Iterative Dataflow Analysis

- ❑ The **Maximum Fixedpoint (MFP)** solution is:
  - Maximum: All values optimistically initialized to  $\top$  values
  - Fixedpoint: Values are propagated until no changes occur
- ❑ MFP is efficiently computed using **worklist** algorithm:

Worklist = all nodes in CFG

while Worklist is not empty:

$n$  = remove a node from Worklist

$OUT[n] = transfer\_function(IN[n])$

    if  $OUT[n]$  changed:

        for each successor  $s$  of  $n$ :

$IN[s] = \wedge (OUT[p] \text{ for } p \text{ in } \text{preds}(s))$

            if  $IN[s]$  changed:

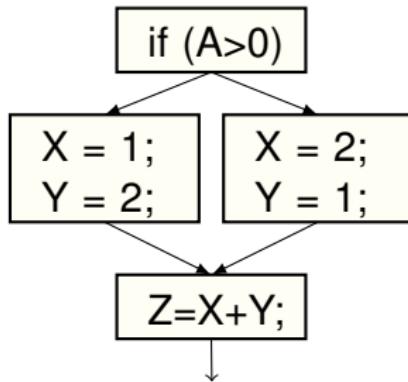
                add  $s$  to Worklist

# Time Complexity of Worklist Algorithm

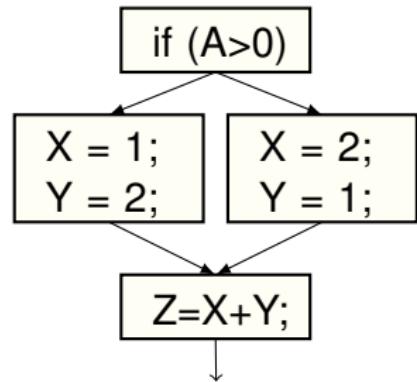
- ❑ Termination: **Greatest lower bound** ensures termination
  - Values start from the top  $\top$  value
  - Values can only flow downward in the semi-lattice
  - Values are guaranteed to reach a fixedpoint in finite steps
- ❑ Time complexity:  $O(d \times (N + E))$ 
  - $d$  = the height of the semi-lattice
  - $N$  = the number of nodes in CFG
  - $E$  = the number of edges in CFG
- ❑ Why?
  1. A node enters worklist only when value changes.
  2. An edge is processed only when value changes.
  3. A value can change at most  $d$  times.

# Maximum Fixedpoint $\leq$ Meet-Over-Paths Solution

## Maximum Fixedpoint (MFP)

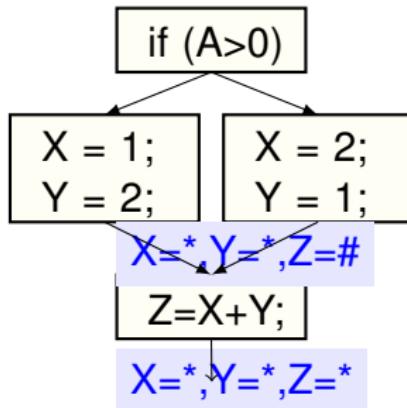


## Meet-Over-Paths (MOP)

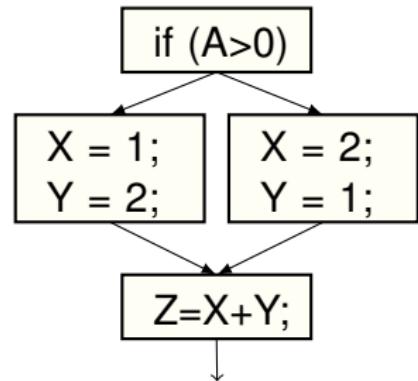


# Maximum Fixedpoint $\leq$ Meet-Over-Paths Solution

## Maximum Fixedpoint (MFP)

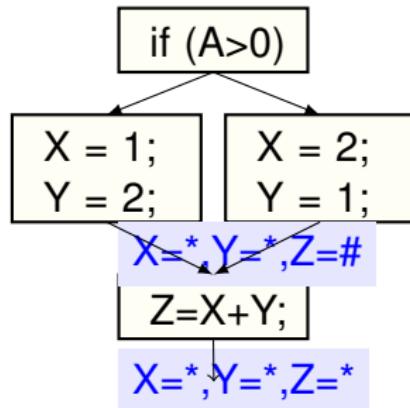


## Meet-Over-Paths (MOP)

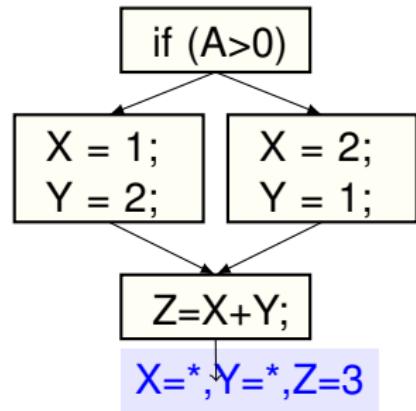


# Maximum Fixedpoint $\leq$ Meet-Over-Paths Solution

## Maximum Fixedpoint (MFP)



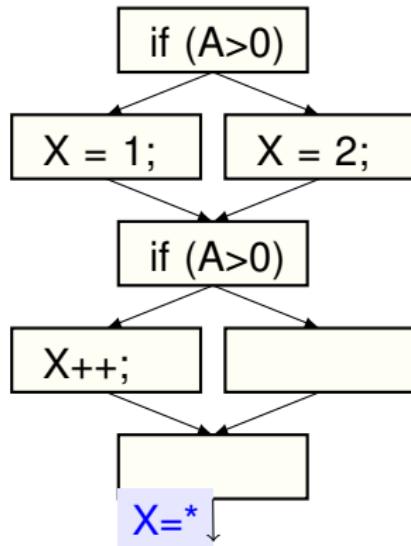
## Meet-Over-Paths (MOP)



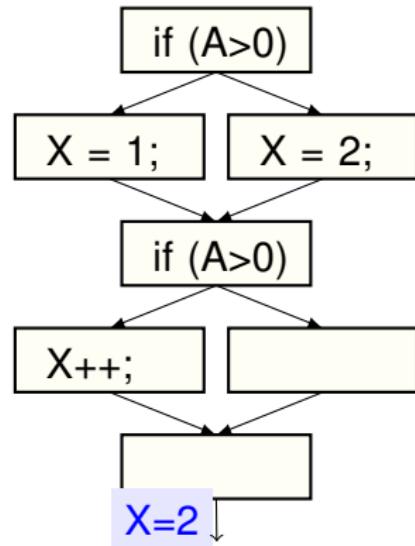
- For MOP,  $OUT[b] = \wedge (OUT[b]$  for all paths to b)
  - Computing  $OUT[b]$  where b is last basic block in example:  
 $OUT[b] = \{X=1, Y=2, Z=3\} \wedge \{X=2, Y=1, Z=3\} = \{X=*, Y=*, Z=3\}$
- MFP  $\leq$  MOP (MFP is less precise)

# Meet-Over-Paths $\leq$ Ideal Solution

## Meet-Over-Paths (MOP)



## Ideal Solution



- ❑ For Ideal,  $OUT[b] = \wedge (OUT[b] \text{ for all } \textit{feasible} \text{ paths to } b)$ 
  - $\text{ideal} = \{X=2\} \wedge \{X=2\} = \{X=2\}$
  - $MOP = \{X=1\} \wedge \{X=2\} \wedge \{X=2\} \wedge \{X=3\} = \{X=*\} \leq \text{Ideal}$

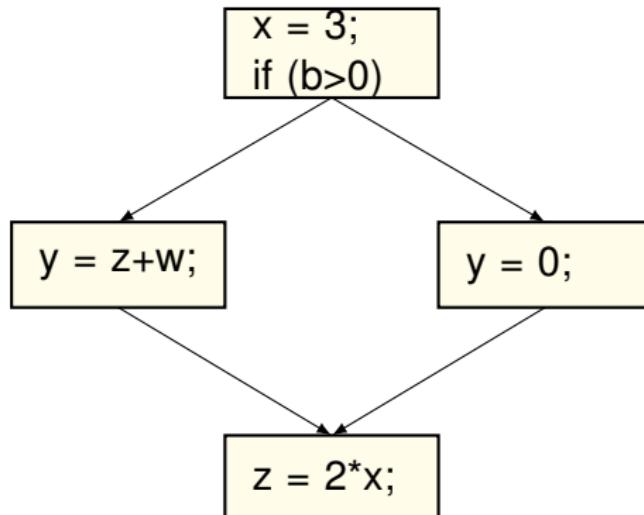
# MFP is Safe but Conservative

- In short, for the GCP dataflow analysis:  
 $\text{Maximum Fixedpoint} \leq \text{Meet-Over-Paths} \leq \text{Ideal}$
- This is both good and bad.
  - Good** : MFP  $\leq$  Ideal means all GCP optimizations are safe.
  - Bad** : MFP  $\leq$  Ideal also means optimizations are conservative.
  
- MOP  $\leq$  Ideal is obvious, but is MFP  $\leq$  MOP true?
- MFP  $\leq$  MOP because GCP is a **monotone framework**.
  - In a monotone framework,  $f(x \wedge y) \leq f(x) \wedge f(y)$   
(Read textbook 9.4.4 for a proof that GCP is monotone.)
- Sometimes MFP = MOP, called **distributive frameworks**.
  - In a distributive framework,  $f(x \wedge y) = f(x) \wedge f(y)$   
(Liveness Analysis, which we'll learn next, is an example.)

# Liveness Analysis

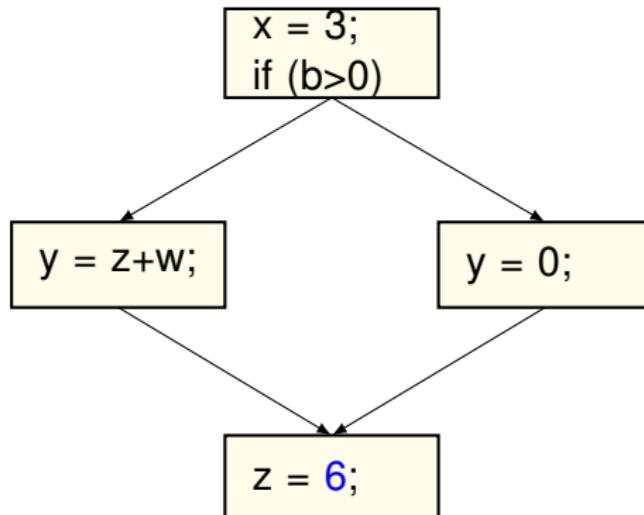
# Another Analysis: Liveness Analysis

- ❑ After GCP, we would like to eliminate the dead code



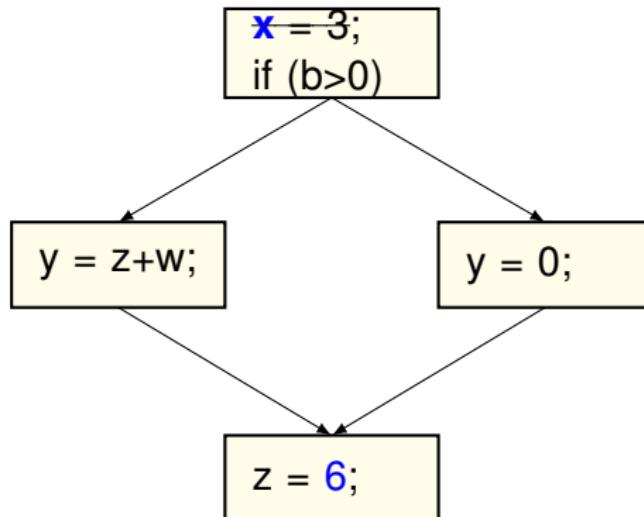
# Another Analysis: Liveness Analysis

- ❑ After GCP, we would like to eliminate the dead code



# Another Analysis: Liveness Analysis

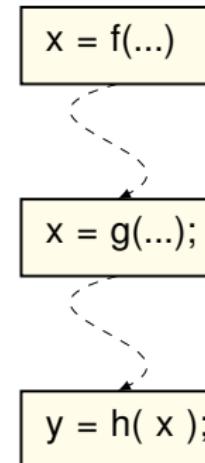
- ❑ After GCP, we would like to eliminate the dead code



# Live/Dead Statement

- A **dead statement** assigns a value that is not used later
- Otherwise, it is a **live statement**

In the example,  
the 1st statement is dead,  
the 2nd statement is live



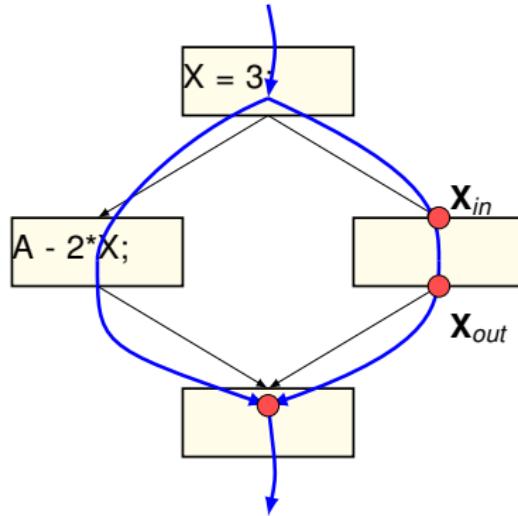
- Assuming inter-procedural analysis says  $f(\dots)$  is internally free of assignments used later (e.g. global variables).

# Global Liveness Analysis (GLA)

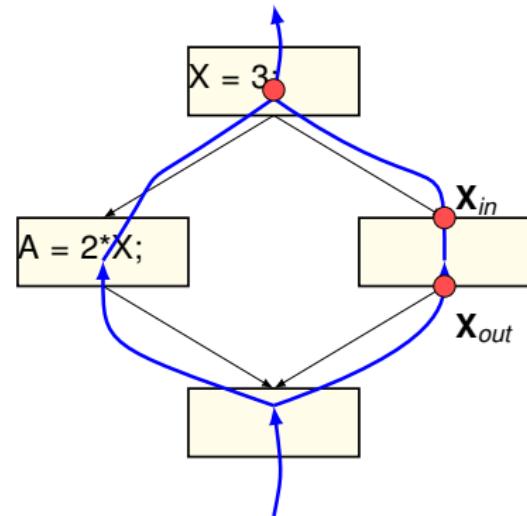
- ❑ Again, let's use the dataflow analysis framework
- ❑ Here are the 4 components of the framework
  - **D**: direction of dataflow for liveness property
  - **V**: domain of values denoting liveness property
  - **$\wedge$** : **meet operator** that merges values when paths meet
  - **F**: **flow propagation function** for liveness
- ❑ This time, liveness property is the set of live variables
  - $\{\}, \{a\}, \{a, b\}, \{b, c\}, \{a, b, c\}, \dots$
- ❑ Meet operator works differently from GCP
  - Meet operator for GCP is an intersection:  
 $x$  is a constant only if  $x$  is same constant along both paths
  - Meet operator for Liveness Analysis is a union:  
 $x$  is live if  $x$  is live along at least one path

# Direction D for GLA

- ❑ Is Liveness a forward or backward analysis?



**Forward Analysis**



**Backward Analysis**

- ❑ Backward, since liveness of a variable flows backward to preceding definitions starting from use

# V and meet operator $\wedge$ for GLA

- ❑ Given variables  $a, b, c$ , domain  $V$  is the set:

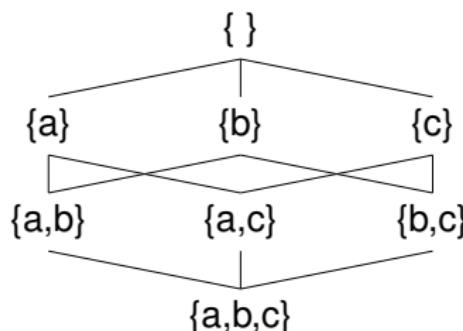
$\{ \}$  /\* no variables are live \*/

$\{a\}, \{b\}, \{c\}$  /\* one variable is live \*/

$\{a,b\}, \{a,c\}, \{b,c\}$  /\* two variables are live \*/

$\{a,b,c\}$  /\* all variables are live \*/

- ❑ Meet operator  $\wedge$  is given by this **semi-lattice**:

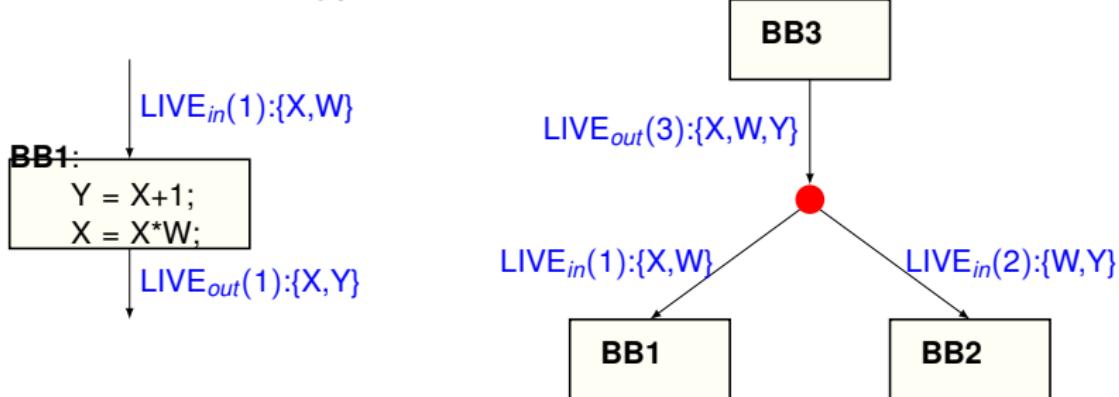


$$\Rightarrow \{a\} \wedge \{b\} = \text{glb}(\{a\}, \{b\}) = \{a,b\}$$

$$\Rightarrow \{b\} \wedge \{a,c\} = \text{glb}(\{b\}, \{a,c\}) = \{a,b,c\}$$

# Dataflow Equations for GLA

- There are two types of flow functions



- Flow transfer function  $F: V \rightarrow V$ 
  - Now  $F$  computes  $P_{in}$  from  $P_{out}$  since it is backward analysis
  - Remove variable definitions, add variable uses to live set
- Meet operator  $\wedge: (V, V) \rightarrow V$ 
  - Merge values from two paths using the previous semi-lattice
  - $LIVE_{out}(i) = \cup LIVE_{in}(k)$  where  $k$  is successor of  $i$

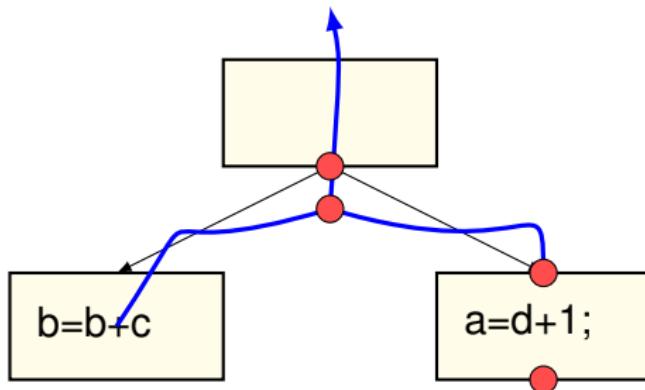
# Flow Transfer Function F for GLA

- ❑ **X(i)**: dataflow property X of basic block i
  - $X_{in}(i)$ : at the entry of basic block i
  - $X_{out}(i)$ : at the exit of basic block i
  
- ❑ F for Global Liveness Analysis (GLA)

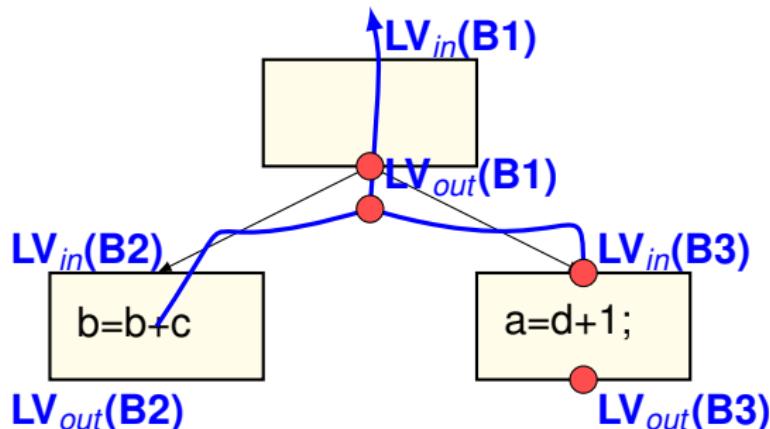
$$\text{LIVE}_{in}(i) = (\text{LIVE}_{out}(i) - \text{DEF}(i)) \cup \text{USE}(i)$$

where  $\text{DEF}(i)$  is the set of defined variables in basic block i  
 $\text{USE}(i)$  is the set of used variables in basic block i

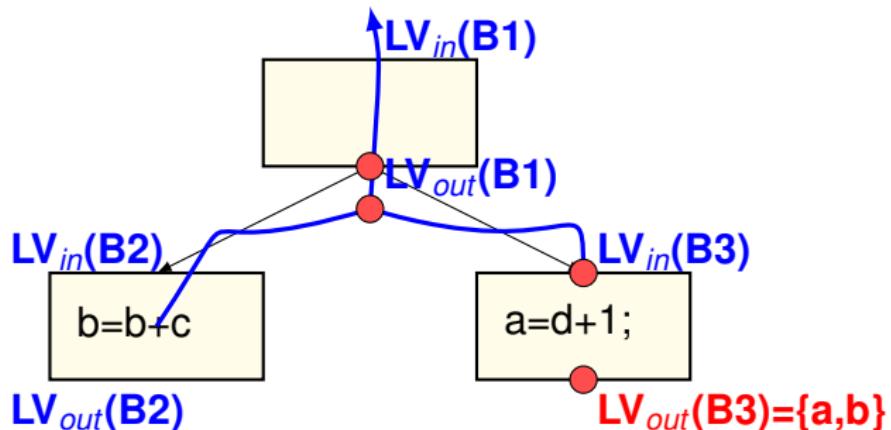
# Liveness Example



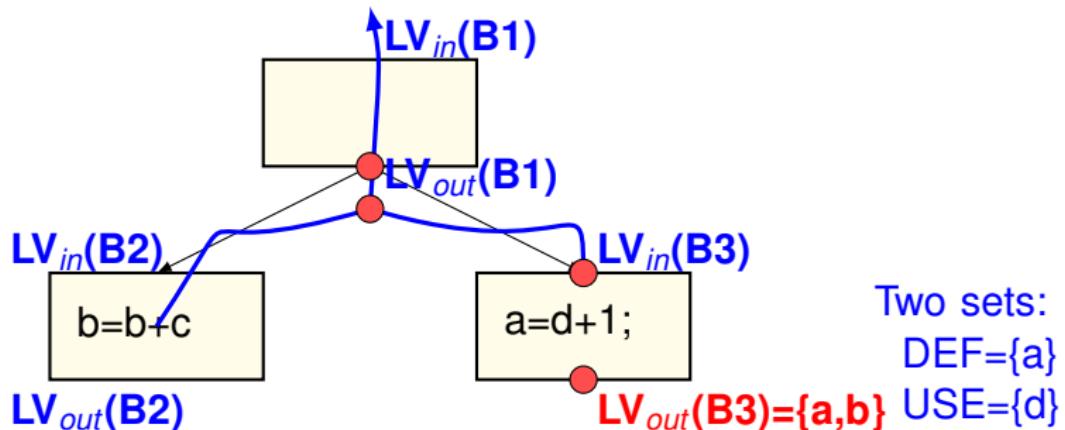
# Liveness Example



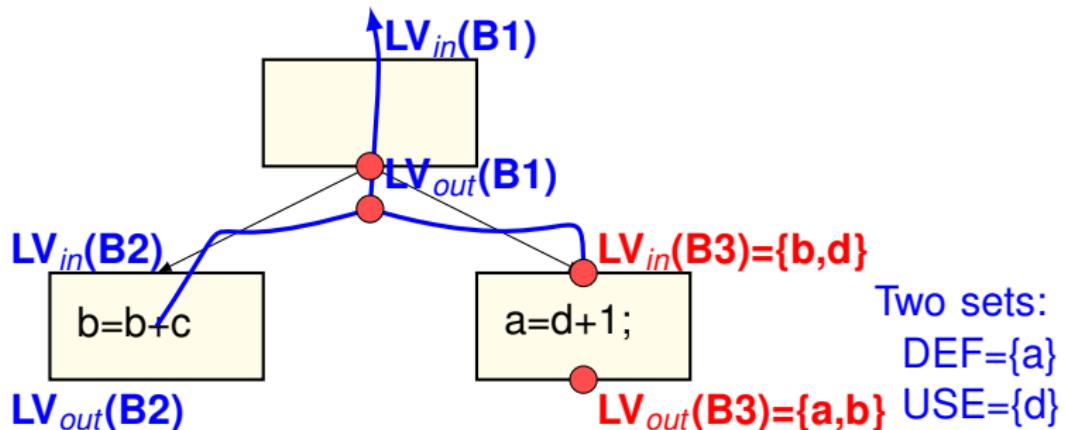
# Liveness Example



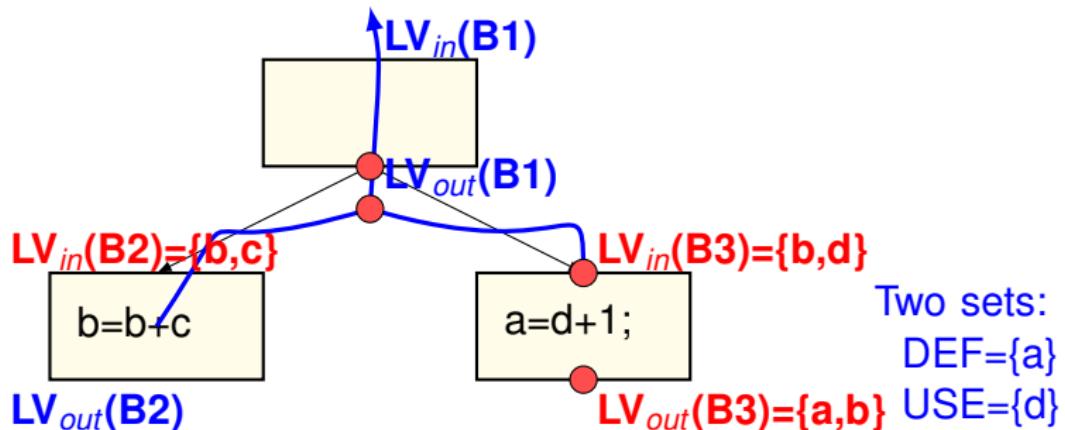
# Liveness Example



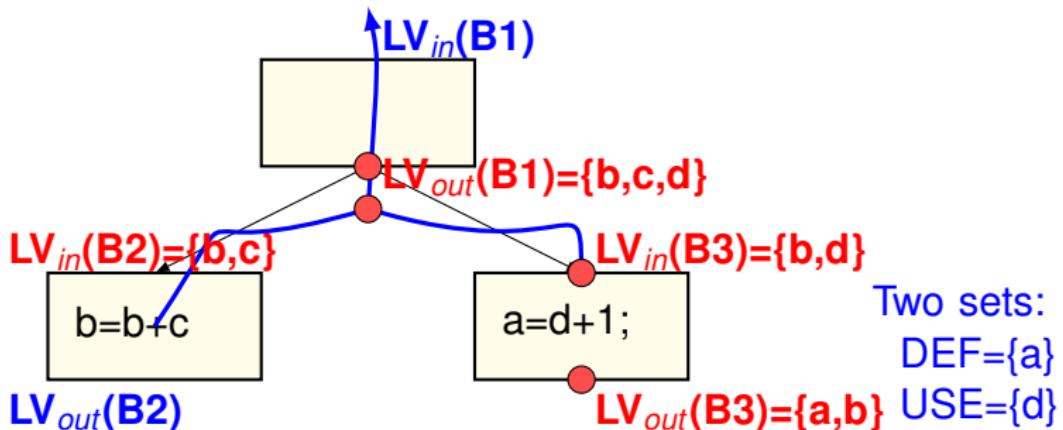
# Liveness Example



# Liveness Example



# Liveness Example

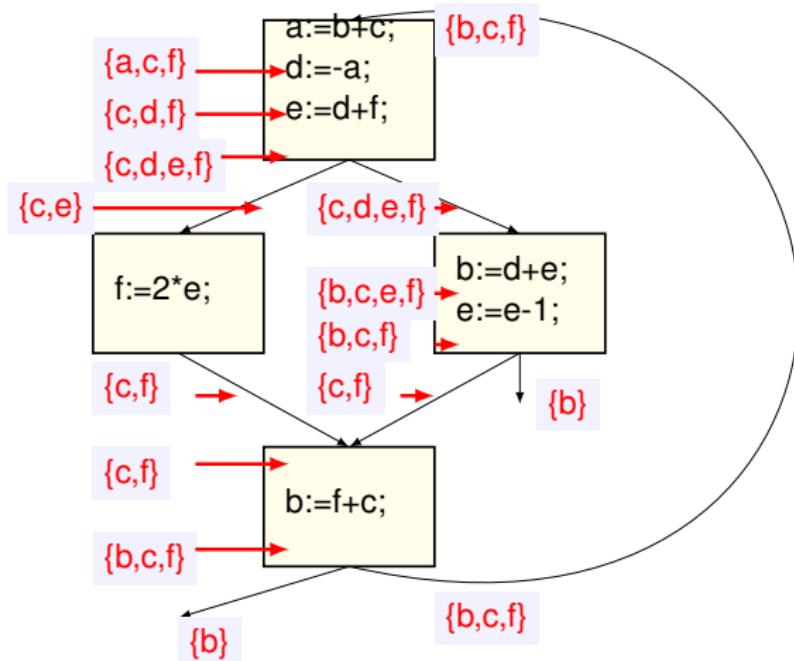


# Applications of Global Liveness Analysis

- ❑ Global Dead Code Elimination is based on GLA
  - A statement  $x = \dots$  is dead code if  $x$  not used
  - Dead statements can be deleted from the program
  
- ❑ Global register allocation is also based on GLA
  - Ideally, all Live variables should be placed in registers
  - If live variables at any point overflow CPU registers,  
some variables have to be stored in stack memory
  - This is called **register spilling**.

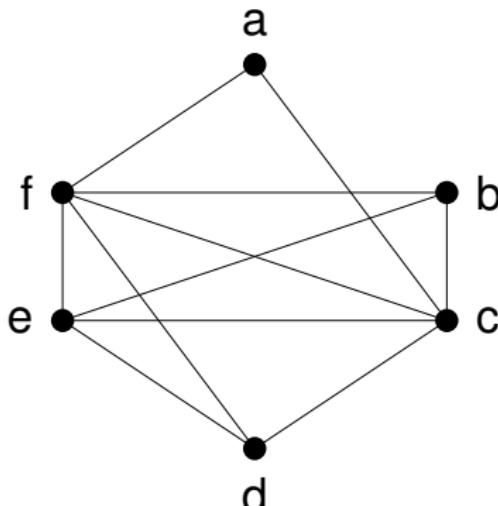
# Register Allocation: Compute Register Interference

- At each point P, compute live variables and interference



# Register Allocation: Register Interference Graph

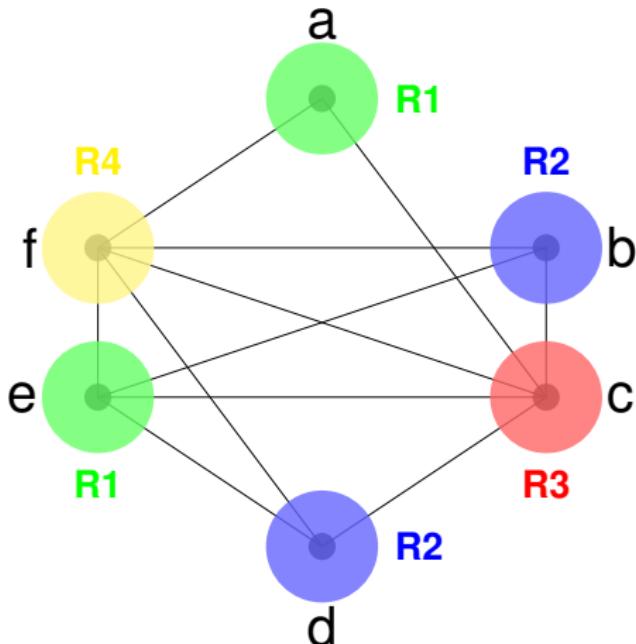
- ❑ Construct **Register Interference Graph (RIG)** such that
  - Nodes represent variables
  - Edges between variables represent interference



- ❑ Two variables can be allocated in same register if no edge
- ❑ Otherwise, they cannot be allocated in the same register

# Register Allocation: Allocation using Graph Coloring

- ❑ Each color represents a CPU register
  - There are 4 colors in the coloring result
  - No register spilling occurs with 4 or more CPU registers



# Summary of Dataflow Analysis

- ❑ A dataflow analysis framework is defined as:  
 $\{ D, V, \wedge: (V, V) \rightarrow V, F: V \rightarrow V \}$ 
  - **D**: direction of dataflow
  - **V**: domain of values denoting property
  - **$\wedge$** : **meet operator** that merges values when paths meet
  - **F**: **flow propagation function** within a basic block
- ❑ Other analyses can be expressed using this framework:
  - Reaching Definitions for Loop Invariant Code Motion (LICM)
  - Available Expressions for Common Subexpression Elimination (CSE)
  - Partial Redundancy Elimination (PRE)
- ❑ Please refer to the textbook on how these are formulated.

# The END !