Topic I (d):

Static Single Assignment Form (SSA)

Reading List

- Slides: Topic Ix
- Other readings as assigned in class

ABET Outcome

- Ability to apply knowledge of SSA technique in compiler optimization
- An ability to formulate and solve the basic SSA construction problem based on the techniques introduced in class.
- Ability to analyze the basic algorithms using SSA form to express and formulate dataflow analysis problem
- A Knowledge on contemporary issues on this topic.

Roadmap

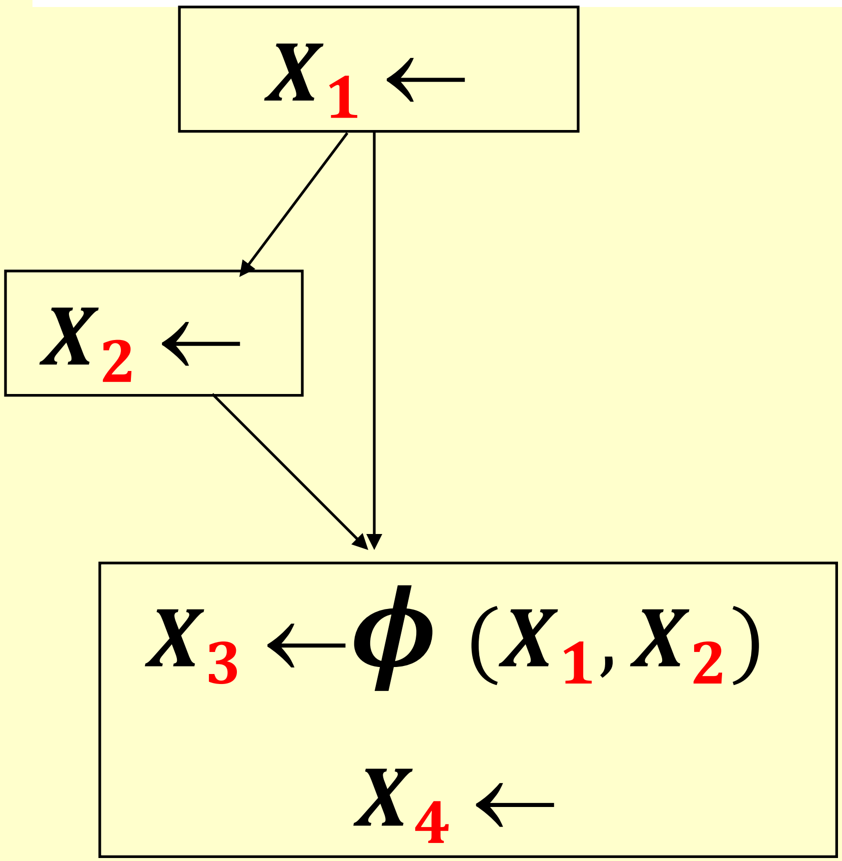
- Motivation
- Introduction:
  - SSA form
  - Construction Method
  - Application of SSA to Dataflow Analysis Problems
- PRE (Partial Redundancy Elimination) and SSAPRE
- Summary

Prelude

- SSA: A program is said to be in SSA form iff
  - Each variable is statically defined exactly only once, and
  - each use of a variable is dominated by that variable's definition.

So, is straight line code in SSA form ?

Example



- In general, how to transform an arbitrary program into SSA form?
- Does the definition of X_2 dominate its use in the example?

SSA: Motivation

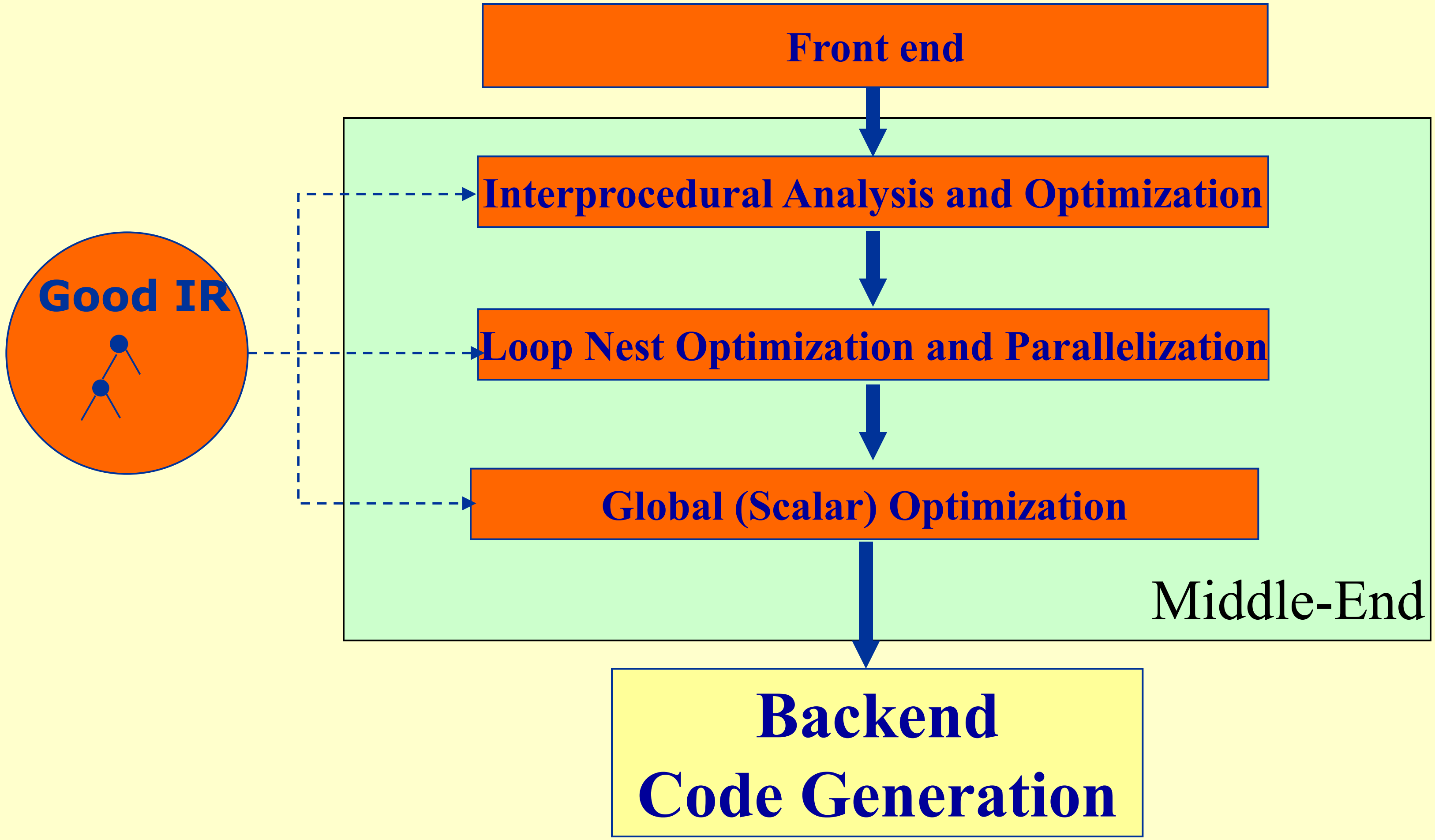
- Provide a uniform basis of an IR to solve a wide range of classical dataflow problems
- Encode both dataflow and control flow information
- A SSA form can be constructed and maintained efficiently
- Many SSA dataflow analysis algorithms are more efficient (have lower complexity) than their CFG counterparts.

Algorithm Complexity

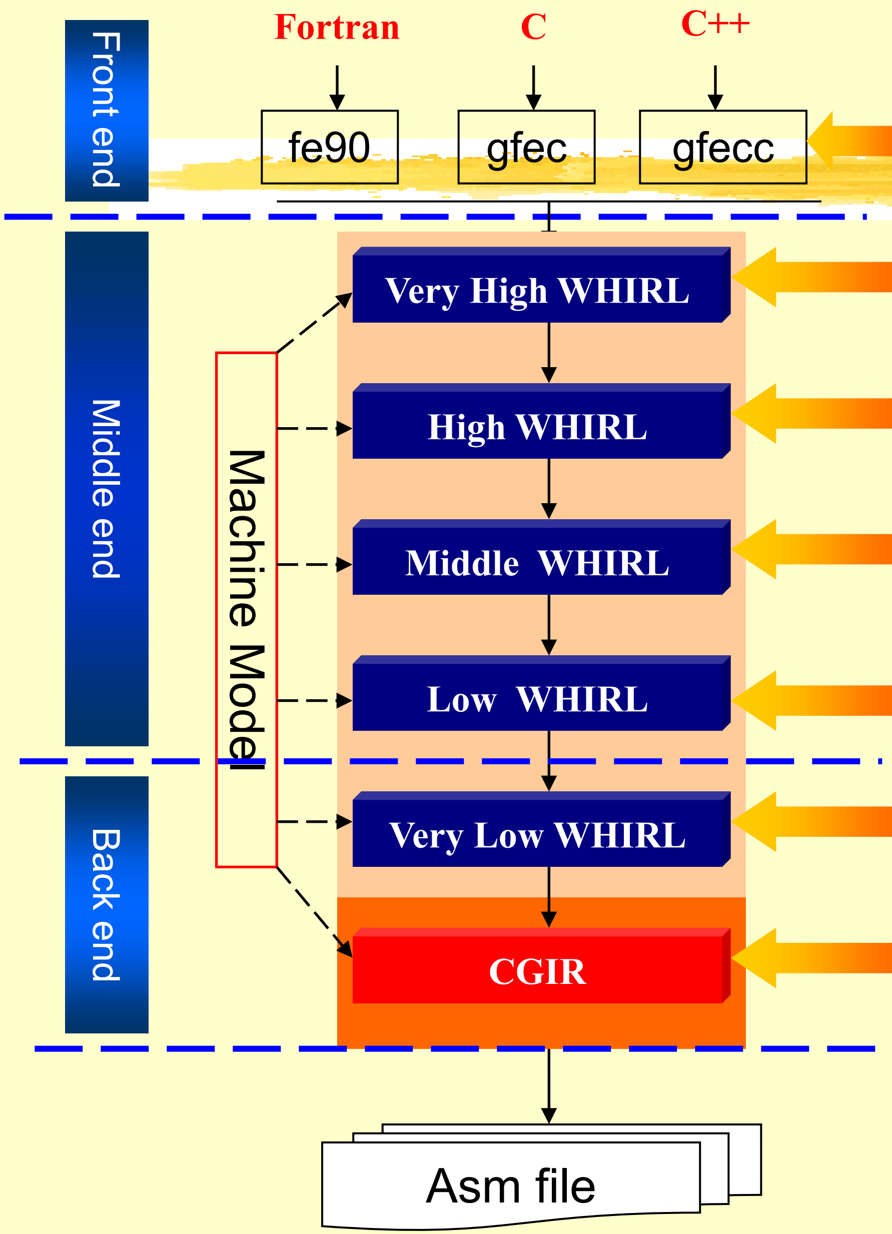
Assume a 1 GHz machine, and
an algorithm that takes f(n) steps (1 step = 1 nanosecond).

n	**8**	**16**	**32**	**128**	**1024**
lg n	3 ns	4 ns	5 ns	7 ns	10 ns
sqrt(n)	2.8 ns	4 ns	6 ns	11 ns	32 ns
n	8 ns	16 ns	32 ns	128 ns	1 μs
n lg n	24 ns	64 ns	160 ns	896 ns	10 μs
n^2	64 ns	256 ns	1.0 μs	16 μs	1 ms
n^3	512 ns	4 μs	32.8 μs	2 ms	1.1 sec
2^n	256 ns	66 μs	4 sec.	10^{22} year	
n!	40 μs	5.8 hours	10^{19} year		

Where SSA Is Used In Modern Compilers ?



KCC Compiler Infrastructure



- Source to IR (Scanner →Parser →RTL →WHIRL)

- VHO (Very High WHIRL Optimizer)
- Standalone Inliner
- W2C/W2F

- IPA (inter-procedural analysis & opt)
- LNO (Loop unrolling/fission/fusion/tiling/peeling etc)
- PREOPT (point-to analysis etc)

- WOPT
  - SSAPRE (Partial Redundancy Elimination)
  - VNFRE (Value numbering based full redundancy elim.)
- RVI-1 (Register Variable Identification)

- RVI-2

- Some peephole opt

- Cflow (control flow opt)
- EBO (extended block opt.)
- PQS (predicate query system)
- Loop Opt (Unrolling + SWP)
- GCM (global code motion), HB sched (hyperblk schedule)
- GRA/LRA (global/local register alloc)

Roadmap

- Motivation
- Introduction:
  - SSA form
  - Construction Method
  - Application of SSA to Dataflow Analysis Problems
- PRE (Partial Redundancy Elimination) and SSAPRE
- Summary

Static Single-Assignment Form

Each variable has only one definition in the program text.

This single *static* definition can be in a loop and may be executed many times. Thus even in a program expressed in SSA, a variable can be dynamically defined many times.

Advantages of SSA

- Simpler dataflow analysis
- No need to use use-def/def-use chains, which requires $N \times M$ space for N uses and M definitions
- SSA form relates in a useful way with dominance structures.

SSA Form – An Example

SSA-form

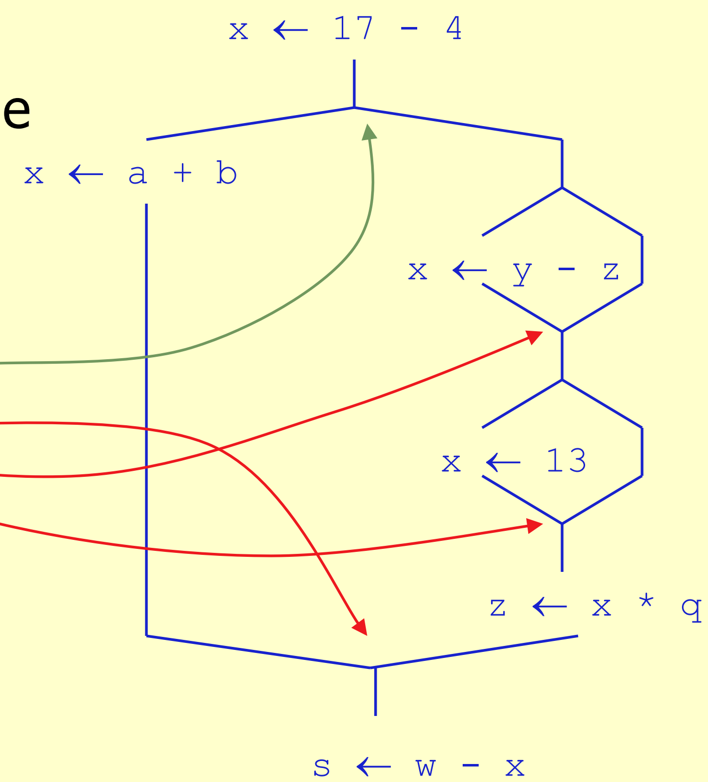
- Each name is defined exactly once
- Each use refers to exactly one name

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

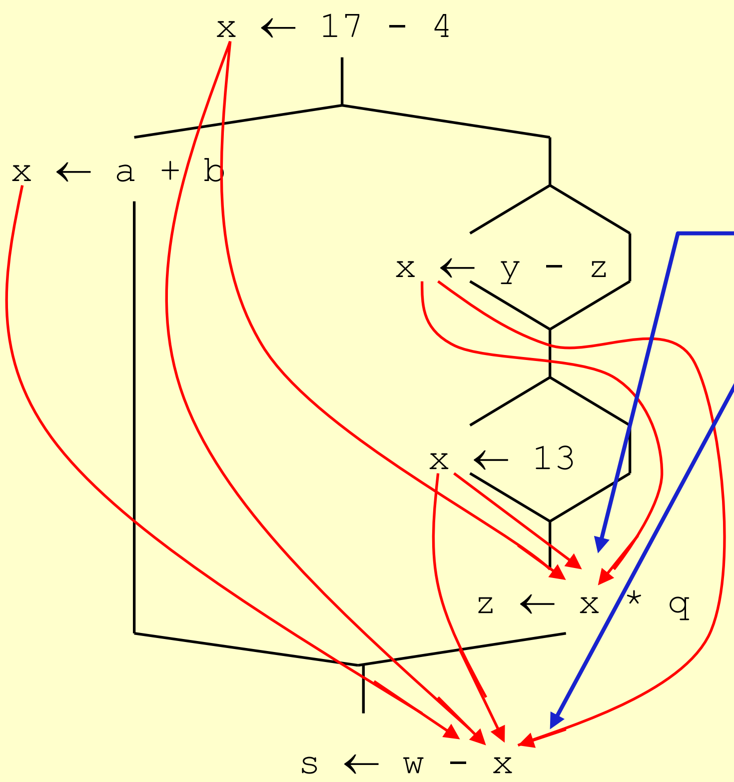


[Curtesy: Slide 10-14 are from the book from Prof. K. Cooper's website]

*

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

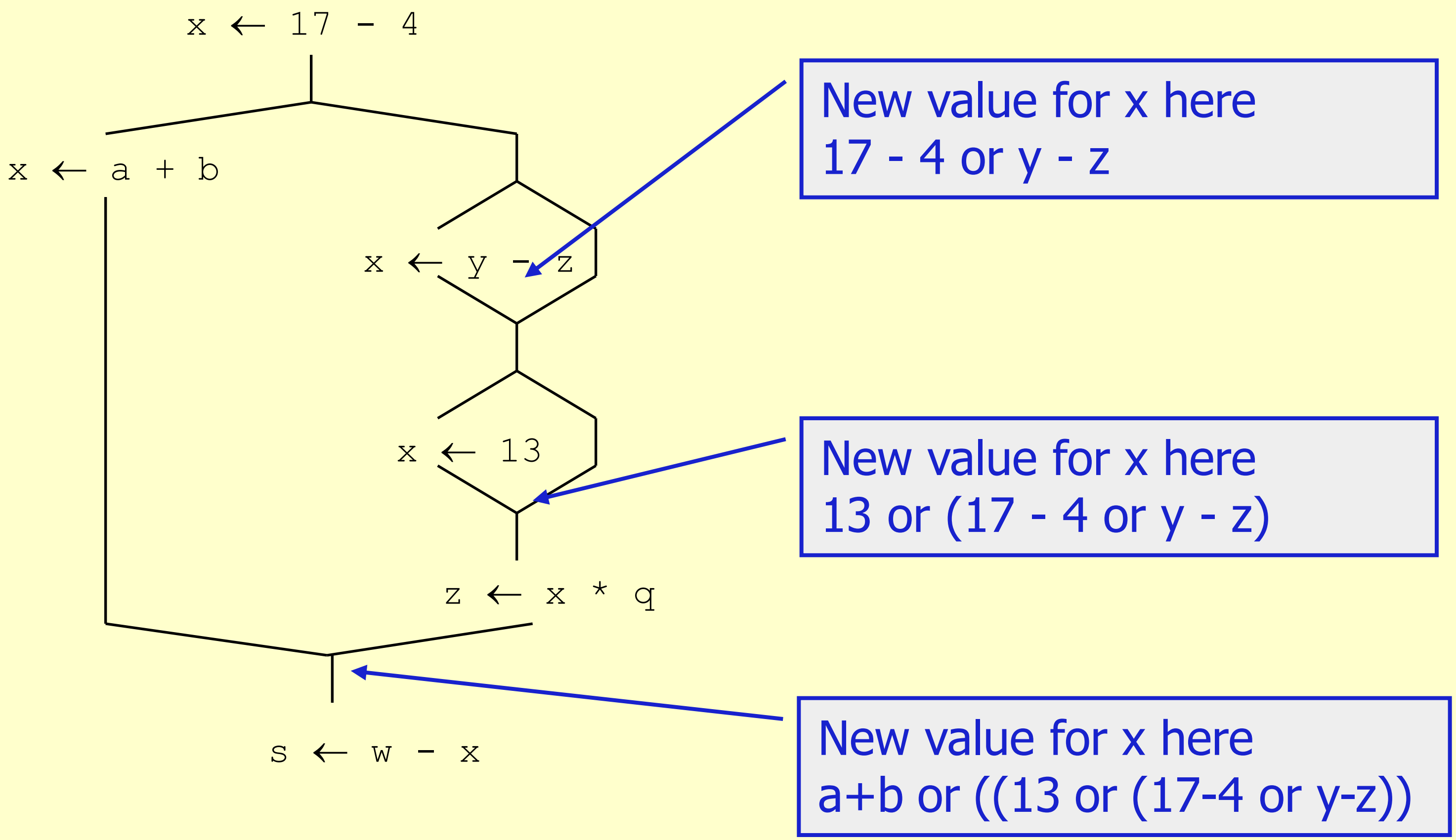
If each value has its own name …

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

*

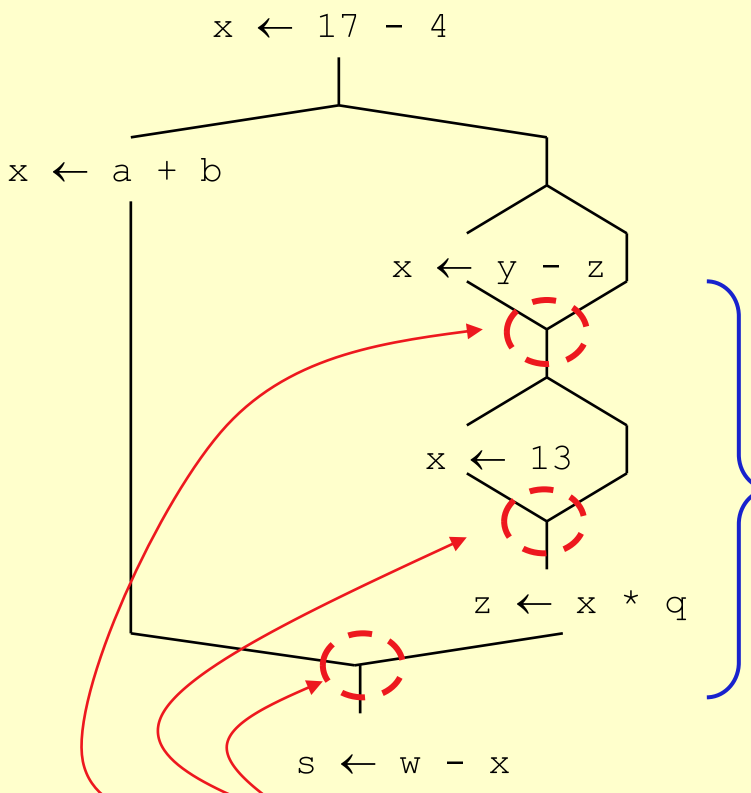
Birth Points (*another notion due to Tarjan*)

Consider the flow of values in this example:



Birth Points (*another notion due to Tarjan*)

Consider the value flow below:



- 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.

$y_1 \leftarrow \ ...$ $\qquad$ $y_2 \leftarrow \ ...$

$y_3 \leftarrow Ø(y_1, y_2)$

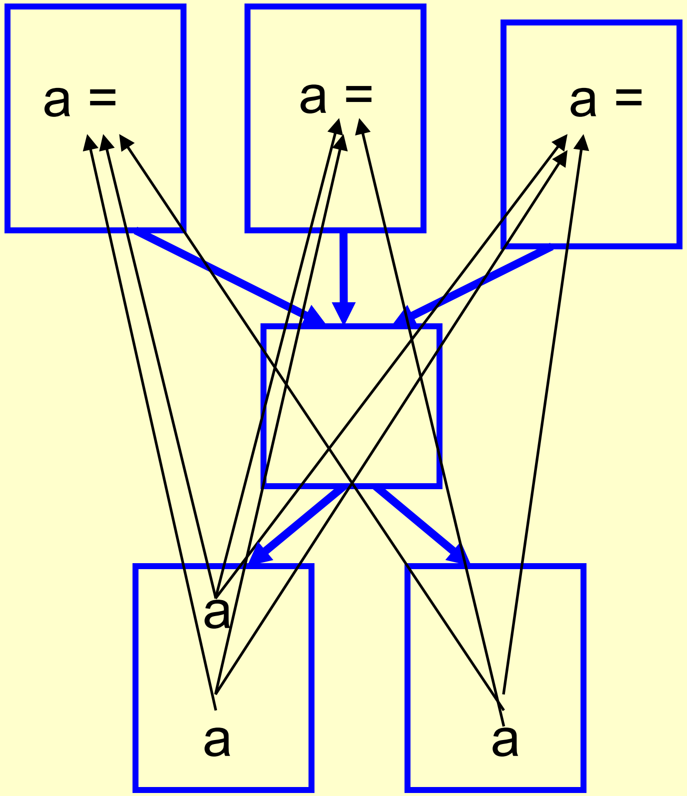
Real machines do not implement a Ø-function directly in hardware.(not yet!)

*

Use-def Dependencies in Non-straight-line Code

Many uses to many defs

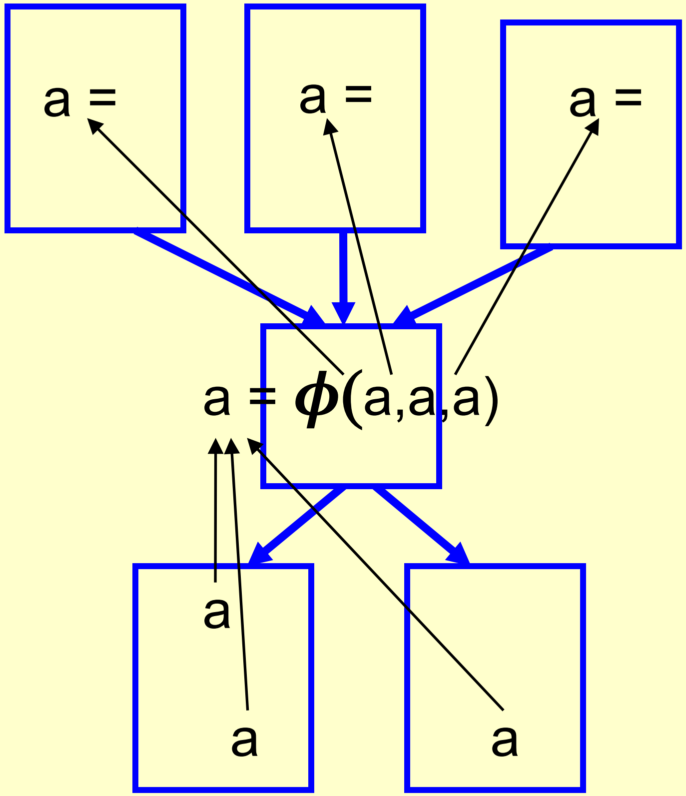
- Overhead in representation
- Hard to manage



Factoring Operator ϕ

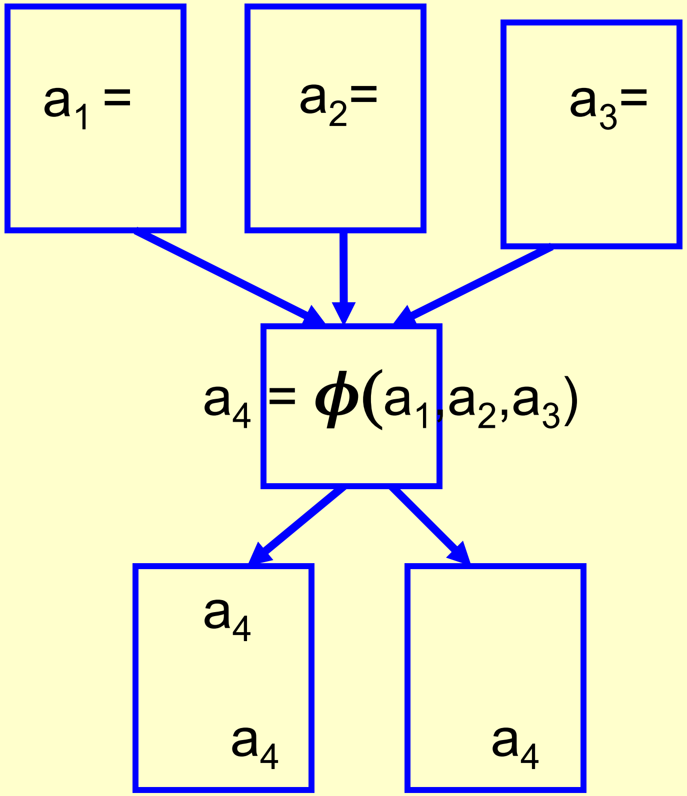
Factoring – when multiple edges cross a *join* point, create a common node Φ that all edges must pass through

- **Number of edges reduced from 9 to 6**
- **A ϕ is regarded as def (its parameters are uses)**
- **Many uses to 1 def**
- **Each def *dominates* all its uses**



Rename to represent use-def edges

- No longer necessary to represent the use-def edges explicitly



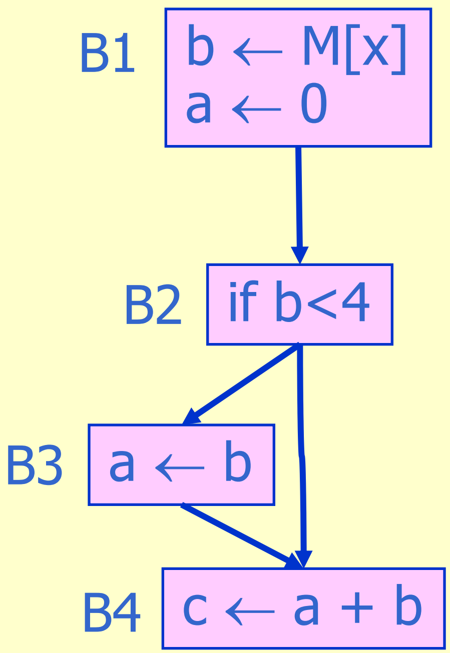
SSA Form in Control-Flow Path Merges

Is this code in SSA form?

No, two definitions of ***a*** at B4 appear in the code (in B1 and B3)

How can we transform this code into a code in SSA form?

We can create two versions of *a*, one for B1 and another for B3.

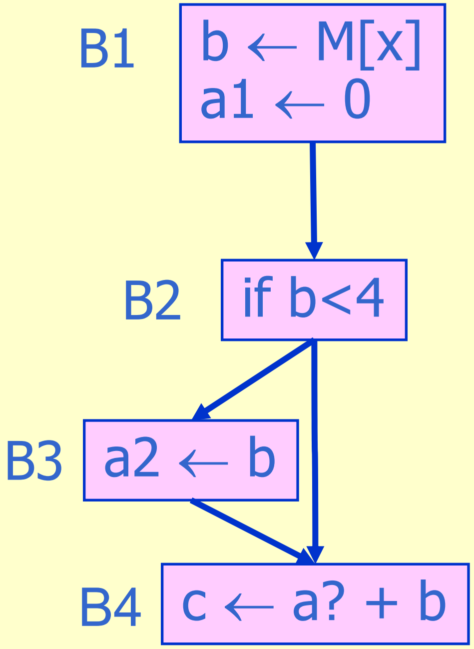


SSA Form in Control-Flow Path Merges

But which version should we use in B4 now?

We define a *fictional* function that "knows" which control path was taken to reach the basic block B4:

$$\phi(a1,a2) = \begin{cases} a1 \text{ if we arrive at B4 from B2} \\ a2 \text{ if we arrive at B4 from B3} \end{cases}$$

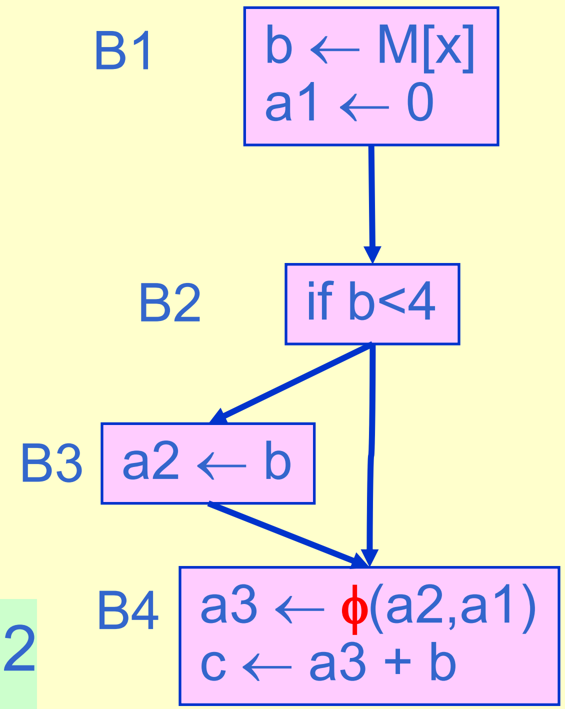


SSA Form in Control-Flow Path Merges

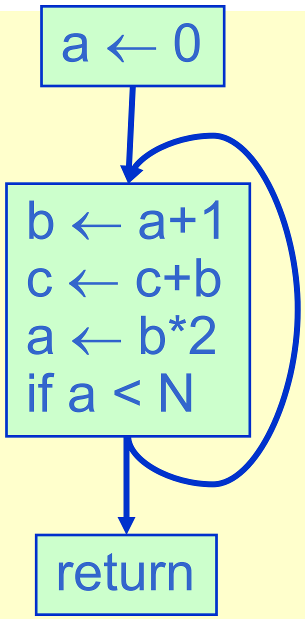
But, which version should we use in B4 now?

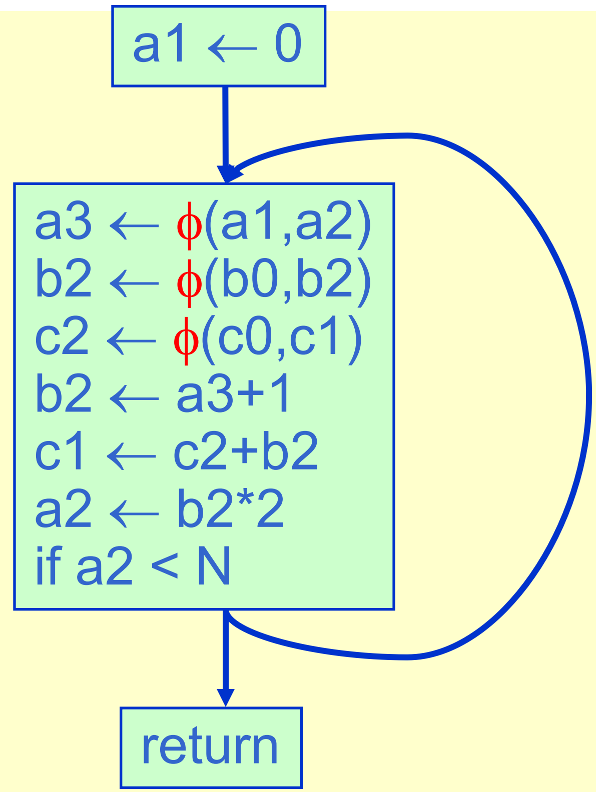
We define a fictional function that "knows" which control path was taken to reach the basic block B4:

$$\phi(a2,a1) = \begin{cases} a1 \text{ if we arrive at B4 from B2} \\ a2 \text{ if we arrive at B4 from B3} \end{cases}$$



A Loop Example





ϕ(b0,b2) is not necessary because b0 is never used. But the phase that generates ϕ functions does not know it. Unnecessary functions are eliminated by dead code elimination.

Note: only a,c are first used in the loop body before it is redefined. For b, it is redefined right at the beginning!

The ϕ function

How can we implement a ϕ function that "knows" which control path was taken?

Answer 1: We don't!! The ϕ function is used only to connect use to definitions during optimization, but is never implemented.

Answer 2: If we must execute the ϕ function, we can implement it by inserting MOVE instructions in all control paths.

Roadmap

- Motivation
- Introduction:
  - SSA form
  - Construction Method
  - Application of SSA to Dataflow Analysis Problems
- PRE (Partial Redundancy Elimination) and SSAPRE
- Summary

Criteria For Inserting ϕ Functions

We could insert one ϕ function for each variable at every *join* point(a point in the CFG with more than one predecessor). But that would be wasteful.

What should be our criteria to insert a ϕ function for a variable *a* at node *z* of the CFG?

Intuitively, we should add a function ϕ if there are two definitions of *a* that can reach the point *z* through distinct paths.

A naïve method

- Simply introduce a ϕ-function at each "join" point in CFG
- But, we already pointed out that this is inefficient – too many useless ϕ-functions may be introduced!
- What is a good algorithm to introduce only the right number of ϕ-functions ?

Path Convergence Criterion

Insert a ϕ function for a variable *a* at a node *z* if all the following conditions are true:

1. There is a block *x* that defines *a*
2. There is a block $y \neq x$ that defines *a*
3. There is a non-empty path *Pxz* from *x* to *z*
4. There is a non-empty path *Pyz* from *y* to *z*
5. Paths *Pxz* and *Pyz* don't have any nodes in common other than *z*
6. ?

The start node contains an implicit definition of every variable.

Iterated Path-Convergence Criterion

The ϕ function itself is a definition of *a*. Therefore the path-convergence criterion is a set of equations that must be satisfied.

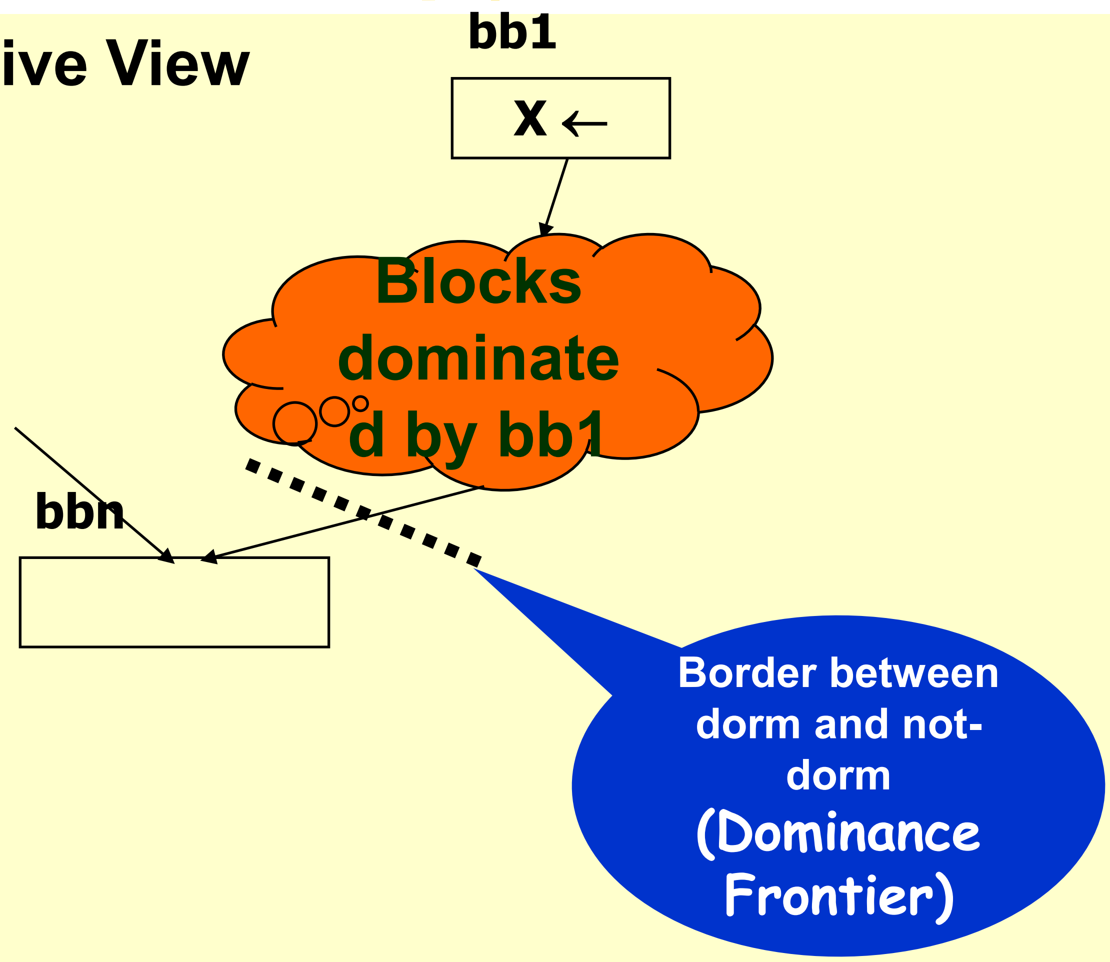
while there are nodes *x*, *y*, *z* satisfying conditions 1-5
and *z* does not contain a ϕ function for *a*
do insert $a \leftarrow \phi(a, a, \ldots, a)$ at node *z*

This algorithm is extremely costly, because it requires the examination of every triple of nodes *x*, *y*, *z* and every path leading from *x* to *y*.

Can we do better? – a topic for more discussion

Concept of dominance Frontiers

An Intuitive View



Dominance Frontier

- The *dominance frontier* DF(x) of a node x is the set of all node z such that x dominates a predecessor of z, without strictly dominating z.

Recall: if x dominates y and x $\neq$ y, then

x *strictly* dominates y

Calculate The Dominance Frontier

An Intuitive Way

How to Determine the Dominance Frontier of Node 5?

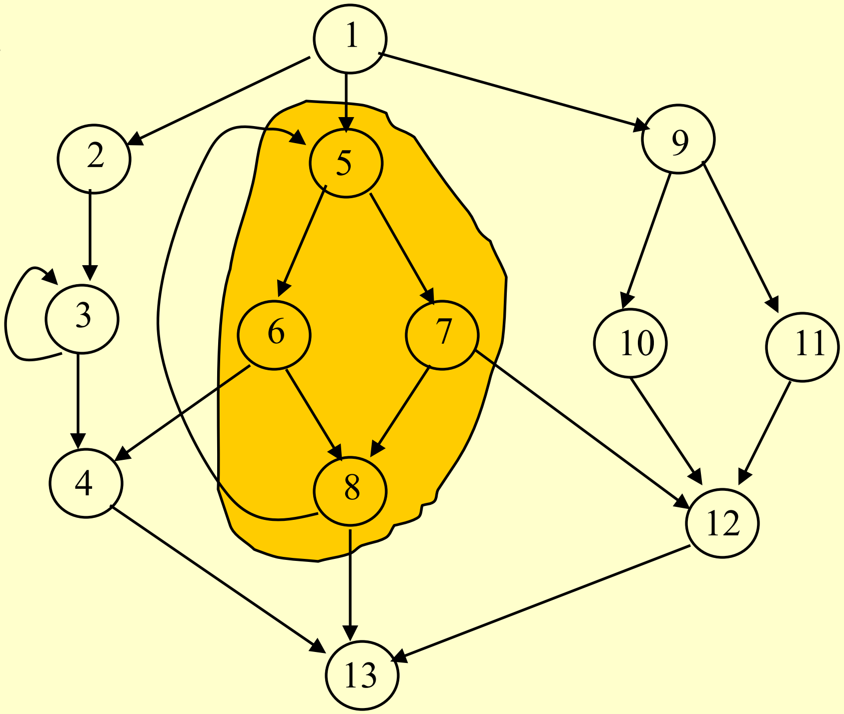
1. Determine the dominance region of node 5:

{5, 6, 7, 8}

2. Determine the targets of edges crossing from the dominance region of node 5

These targets are the dominance frontier of node 5

DF(5) = { 4, 5, 12, 13}



NOTE: node 5 is in DF(5) in this case – why ?

Are we done ?

- Not yet!
- See a simple example ..

Putting program into SSA form

- Φ needed only at *dominance frontiers* of defs (where it *stops* dominating)
- Dominance frontiers pre-computed based on control flow graph
- Two phases:
  1. Insert Φ's at dominance frontiers of each def (recursive)
  2. Rename the uses to their defs' name
     - Maintain and update stack of variable versions in <u>pre-order</u> traversal of dominator tree

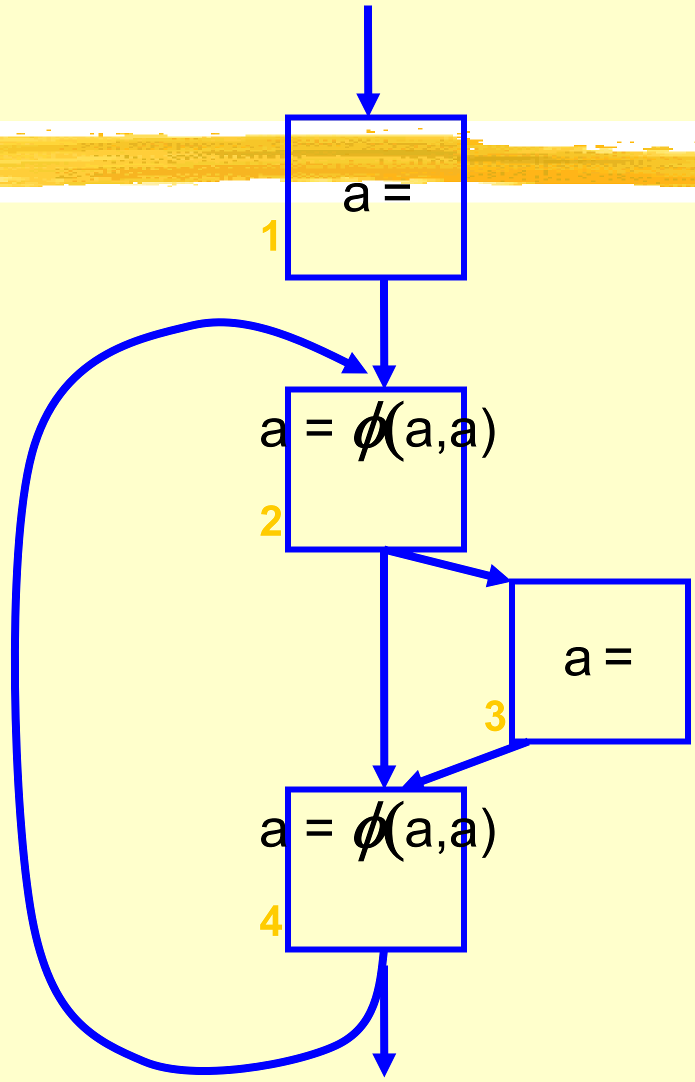
Example

Phase 1: Φ Insertion

Steps:

def at BB 3 → Φ at BB 4

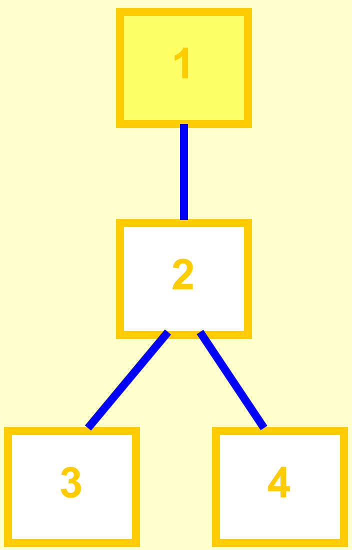
Φ def at BB 4 → Φ at BB 2

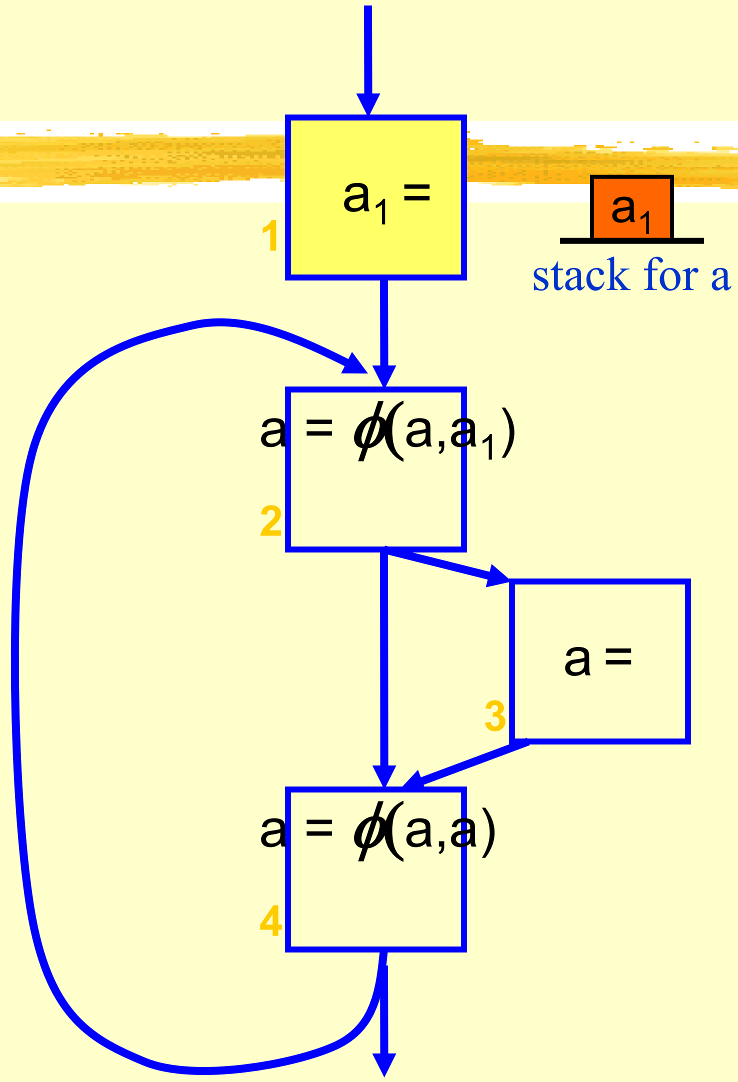


Example

Phase 2: Rename

dominator tree

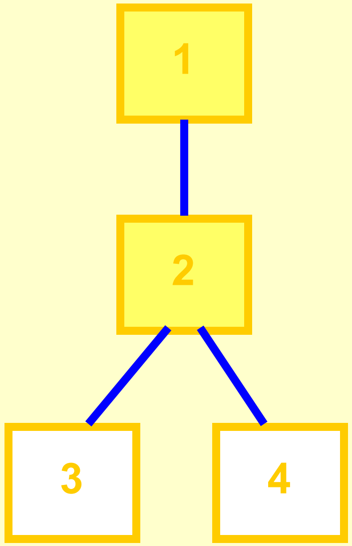


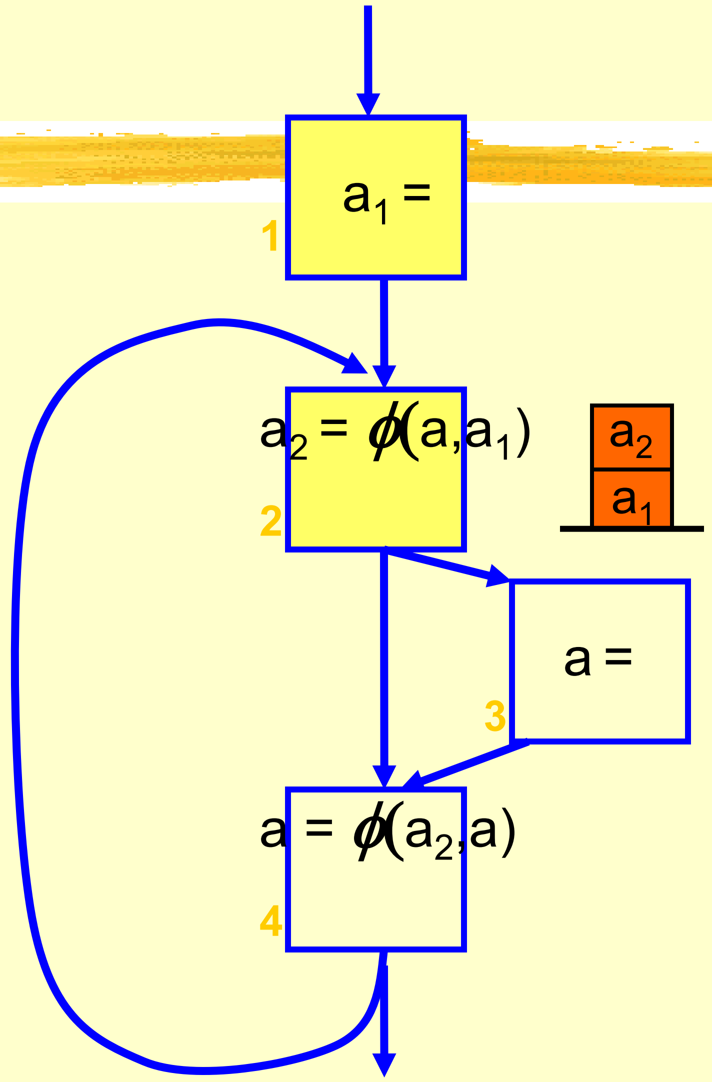


Example

Phase 2: Rename

dominator tree

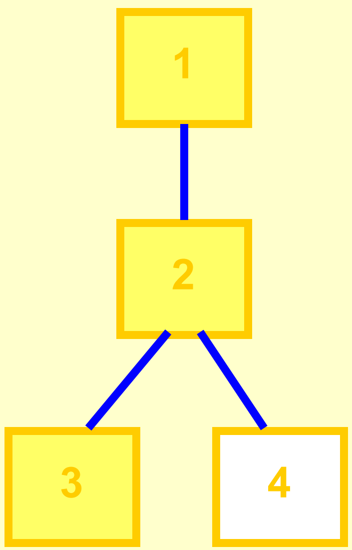


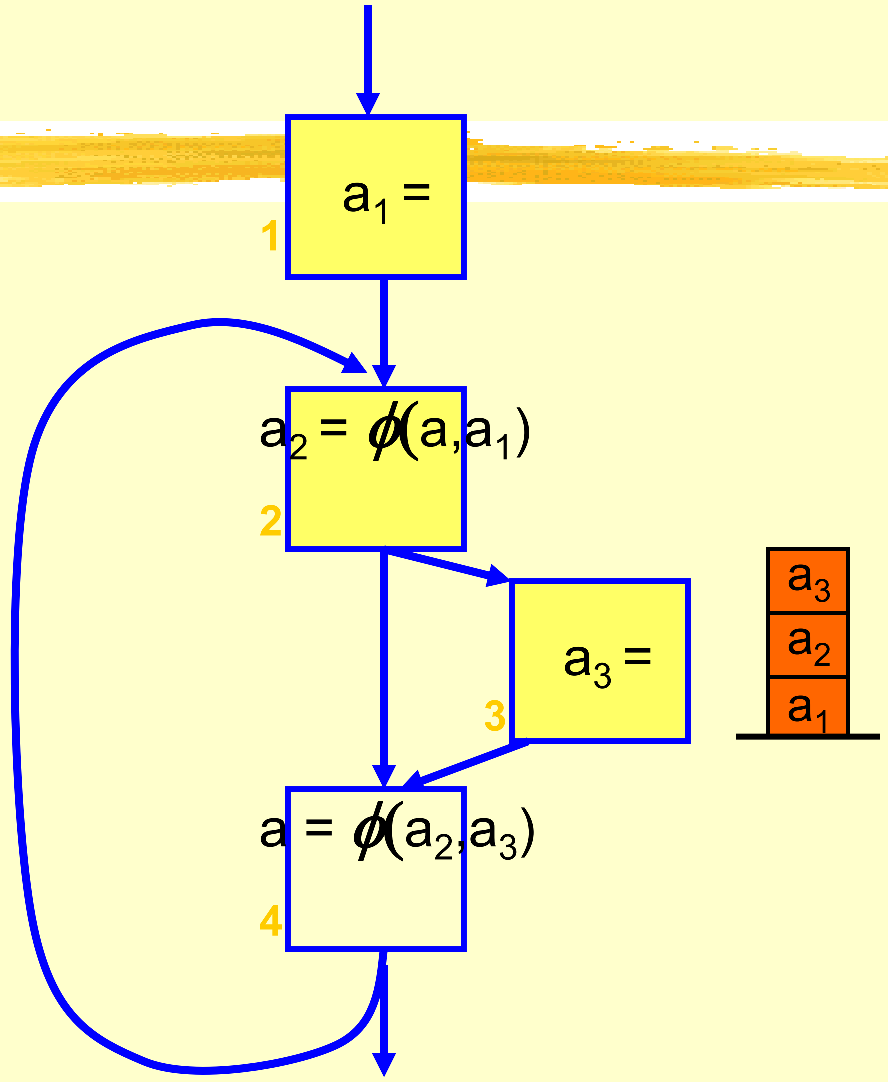


Example

Phase 2: Rename

dominator tree

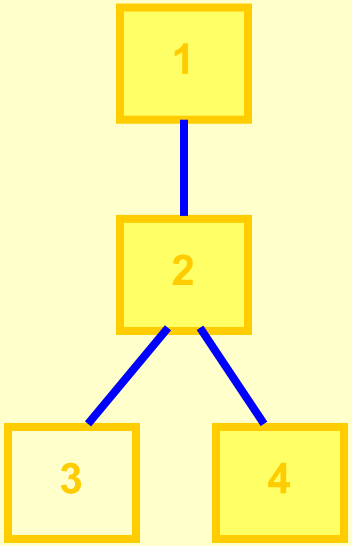


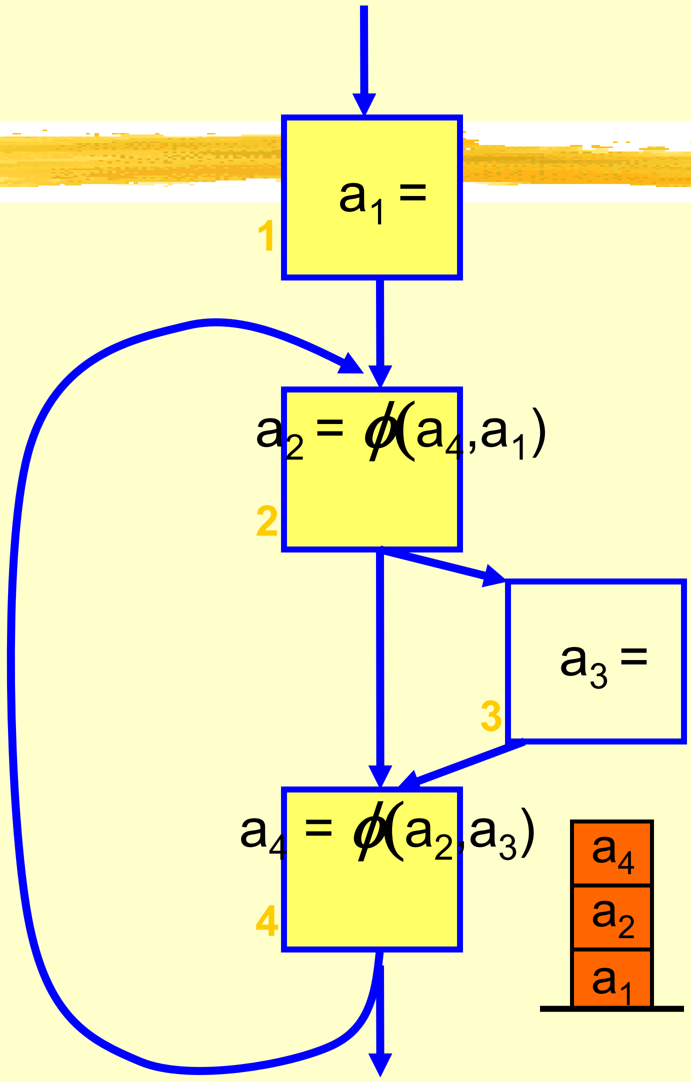


Example

Phase 2: Rename

dominator tree





Roadmap

- Motivation
- Introduction:
  - SSA form
  - Construction Method
  - Application of SSA to Dataflow Analysis Problems
- PRE (Partial Redundancy Elimination) and SSAPRE
- Summary

Simple Constant Propagation in SSA

If there is a statement *v* ← *c,* where *c* is a constant, then all uses of *v* can be replaced for *c*.

A ϕ function of the form *v* ← ϕ(c1, c2, …, cn) where all ci's are identical can be replaced for *v* ← *c.*

Using a work list algorithm in a program in SSA form, we can perform constant propagation in linear time

In the next slide we assume that *x*, *y*, *z* are variables and *a*, *b*, *c* are constants.

Linear Time Optimizations in SSA form

Copy propagation: The statement $x \leftarrow \phi(y)$ or the statement $x \leftarrow y$ can be deleted and y can substitute every use of x.

Constant folding: If we have the statement $x \leftarrow a \oplus b$, we can evaluate $c \leftarrow a \oplus b$ at compile time and replace the statement for $x \leftarrow c$

Constant conditions: The conditional

if $a < b$ goto L1 else L2

can be replaced for goto L1 or goto L2, according to the compile time evaluation of $a < b$, and the CFG, use lists, adjust accordingly

Unreachable Code: eliminate unreachable blocks.

Dead-Code Elimination in SSA Form

Because there is only one definition for each variable, if the list of uses of the variable is empty, the definition is dead.

When a statement $v \leftarrow x \oplus y$ is eliminated because v is dead, this statement should be removed from the list of uses of x and y. Which might cause those definitions to become dead.

Thus we need to iterate the dead code elimination algorithm.

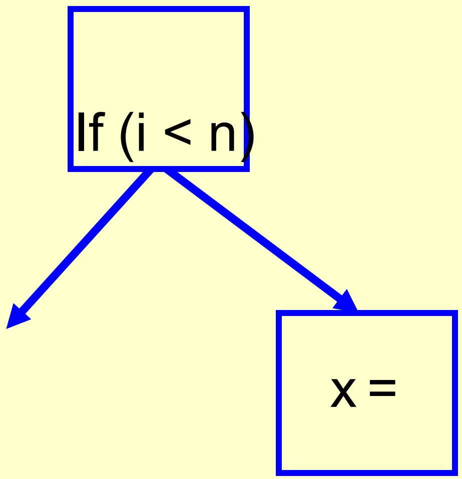
A Case Study: Dead Store Elimination

Steps:

1. Assume all defs are dead and all statements *not* required
2. Mark following statements <u>required:</u>
   a. Function return values
   b. Statements with side effects
   c. Def of global variables
3. Variables in required statements are *live*
4. Propagate liveness backwards iteratively through:
   a. use-def edges – when a variable is live, its def statement is made live
   b. control dependences

Control Dependence

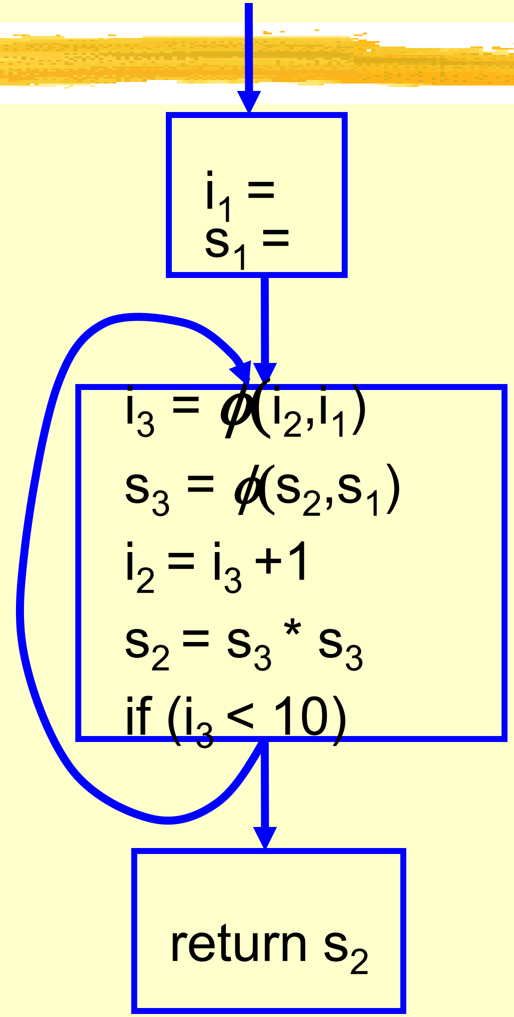
- Statements in branched-to blocks depend on the conditional branch
- Equivalent to post-dominance frontier (dominance frontier of the inverted control flow graph)



Example of dead store elimination

Propagation steps:

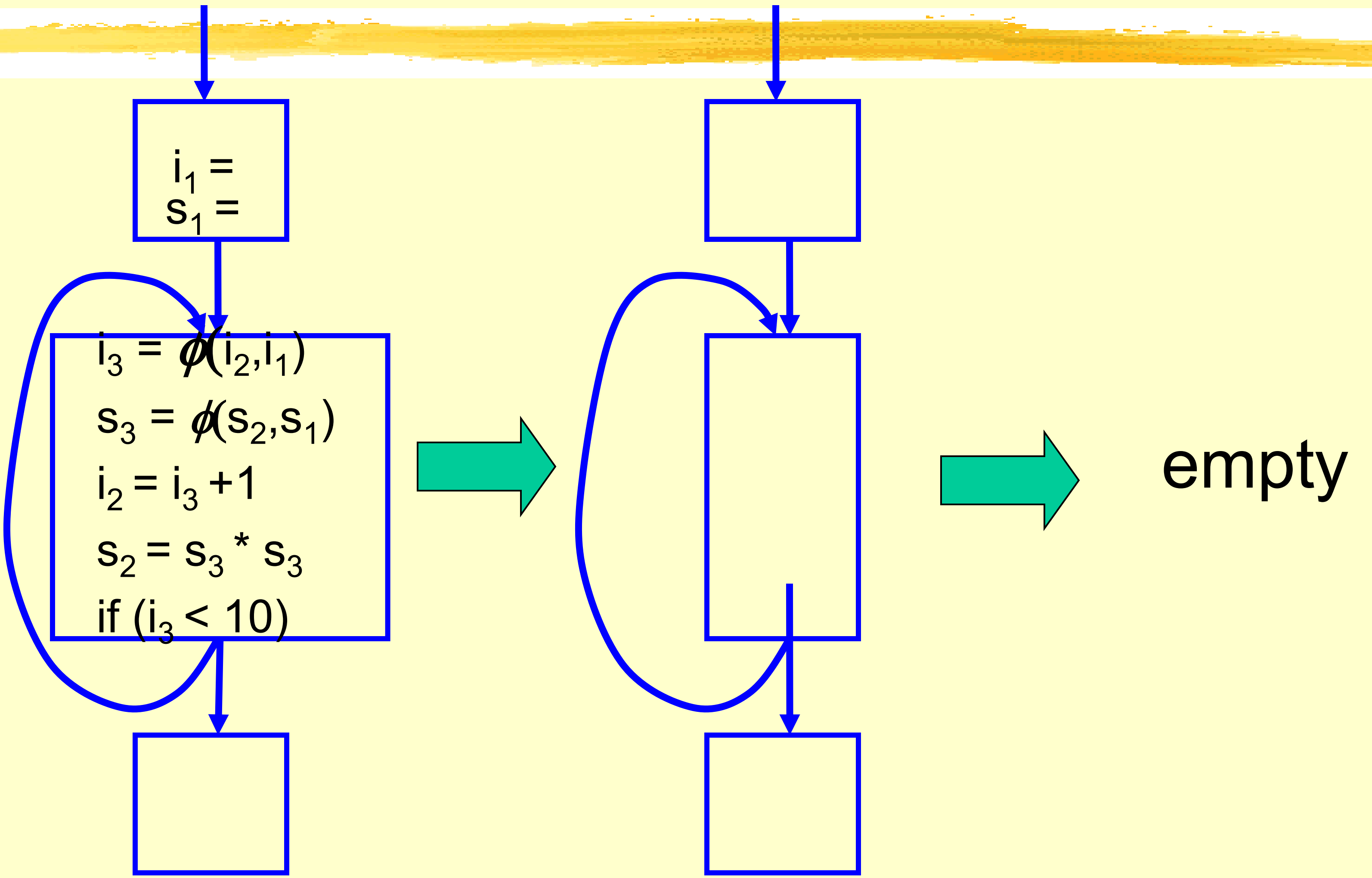
1. return $s_2 \longrightarrow s_2$
2. $s_2 \longrightarrow s_2 = s_3 * s_3$
3. $s_3 \longrightarrow s_3 = \phi(s_2, s_1)$
4. $s_1 \longrightarrow s_1 =$
5. return $s_2 \rightarrow$ if $(i_2 < 10)$ [control dependence]
6. $i_2 \longrightarrow i_2 = i_3 + 1$
7. $i_3 \longrightarrow i_3 = \phi(i_2, i_1)$
8. $i_1 \longrightarrow i_1 =$



Nothing is dead

Example of dead store elimination

All statements not required; whole loop deleted



Advantages of SSA-based optimizations

Dependency information built-in

- No separate phase required to compute dependency information

Transformed output preserves SSA form

- Little overhead to update dependencies

Efficient algorithms due to:

- Sparse occurrence of nodes
  - Complexity dependent only on problem size (independent of program size)
- Linear data flow propagation along use-def edges
- Can customize treatment according to candidate

Can re-apply algorithms as often as needed

No separation of local optimizations from global optimizations