Static Single Assignment (SSA) Objectives

- Given a CFG, the student will be able to compute the dominator relation for the CFG.
- Given a CFG, the student will be able to compute the dominance frontier and iterated dominance frontier for each node in the CFG.
- Given a CFG, the student will be able to compute the SSA-form for the CFG.
- Given SSA-form, the student will be able to convert it to normal form.

SSA Form

- An improvement to DU-UD chains.
- sparse representation
- each variable is defined once → each use has one reaching definition
- use ϕ-nodes to merge multiple definitions reaching a single point

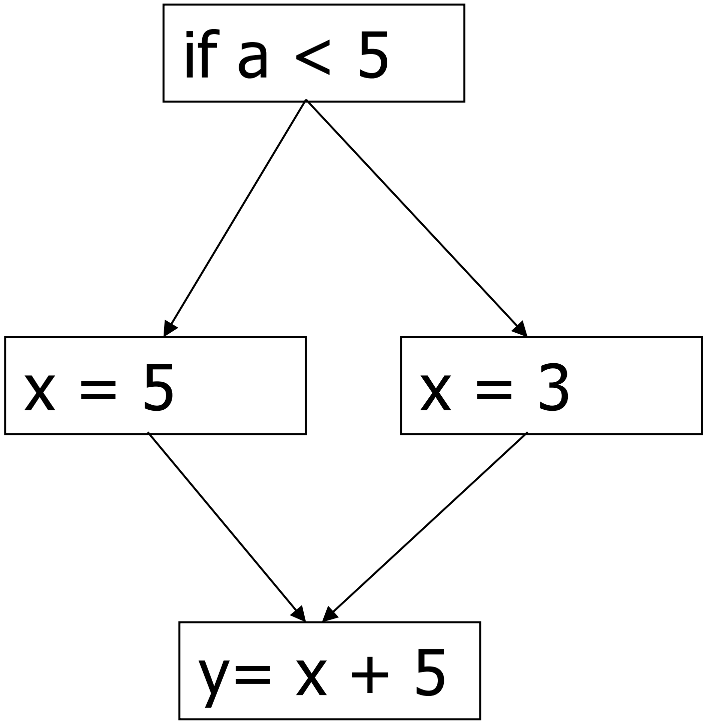
$v = 4$
$x = v + 5$
$v = 6$
$y = v + 7$

becomes

$v_0 = 4$
$x_0 = v_0 + 5$
$v_1 = 6$
$y_0 = v_1 + 7$

Control Flow

- What do we do when there are multiple definitions reaching a single point?
- In the example to the right, what if the definition of x used at in the computation of y?

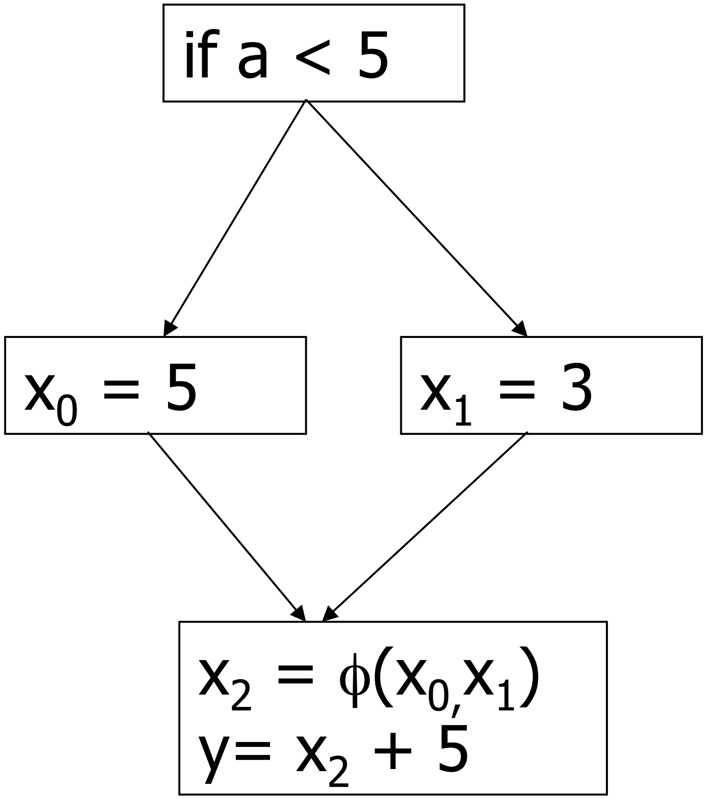


ϕ-nodes

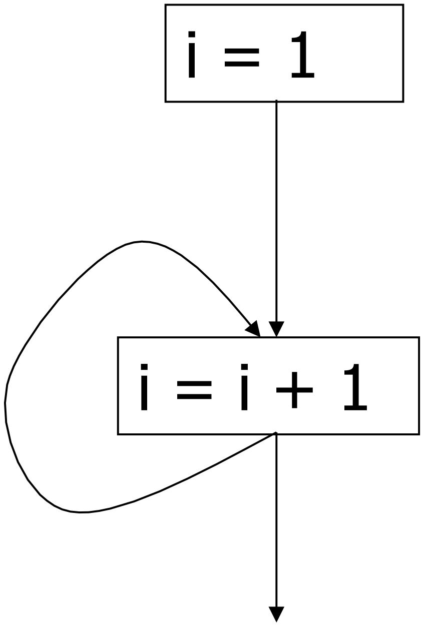
- Def^n: Consider a block b in the CFG with predecessors $\{p_1, p_2, \ldots, p_n\}$ where $n > 1$. A ϕ-node

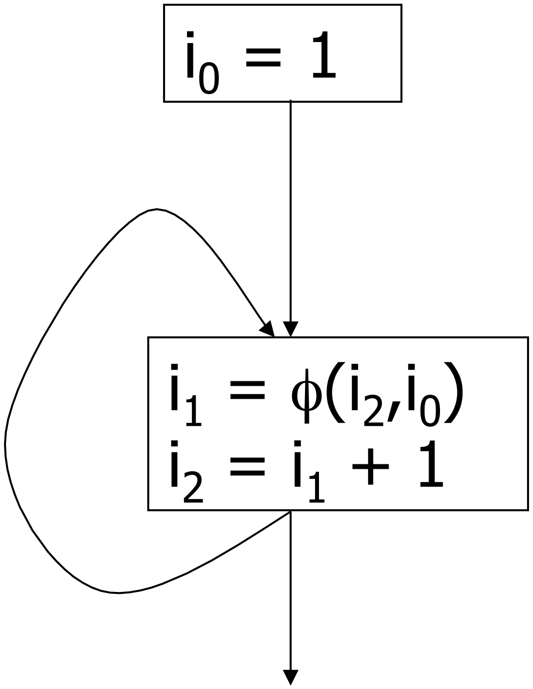
  $T_0 = \phi(T_1, T_2, \ldots, T_n)$

  in b gives the value of T_i to T_0 on entry to b if the execution path leading to b has p_i as the predecessor to b.



Another Example





Placing ϕ-nodes

- Find the join points
  - top of basic blocks where different definitions reach on different paths
- Method
  - computing dominator relation for CFG
  - compute dominance frontiers for each basic block

Dominator Relation

- Def^n: A node n in a graph dominates a node m, denoted $n \underline{\gg} m$, if every path from the entry node to m contains n.

  $n \underline{\gg} n$ (reflexive)

  $n \underline{\gg} m \land n \neq m \rightarrow !(m \underline{\gg} n)$ (antisymmetric)

  $n \underline{\gg} m \land m \underline{\gg} r \rightarrow n \gg r$ (transitive)

- $\underline{\gg}$ is a partial order on the CFG nodes

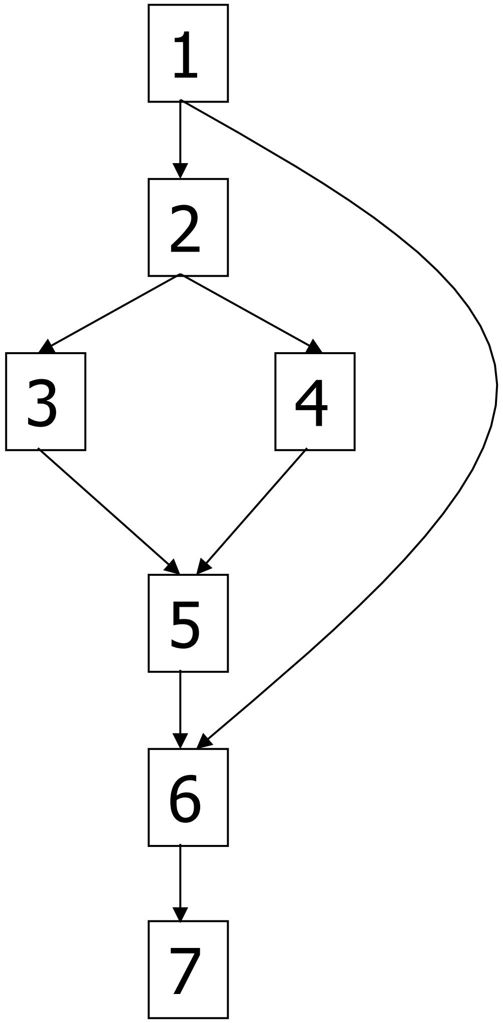
Computing Dominators

```
$D(v_0) = \{v_0\}$
for each $n \in V - \{v_0\}$ //init
    $D(n) = V$
do {
    for each $n \in V - \{v_0\}$
      $D(n) = \{n\} \cup$
                  $\bigcap_{p \in preds(n)} D(p)$
} until no D(n) changes
```

$n \underline{\gg} m \Leftrightarrow n=m$ *or* $\forall p \in pred(m)\ n \underline{\gg} p$

- ENTRY dominates all nodes
- Since $\underline{\gg}$ is a partial order, we can construct an ordering of all the nodes that each node dominates to construct a dominator tree.
- The immediate dominator of n, denoted idom(n), is its parent in the dominator tree.

Example



Dominance Frontiers

- Def^n: Node n is said to strictly dominate a node m, denoted $n \gg m$, if $n \neq m \wedge n \underline{\gg} m$.
- Def^n: The dominance frontier of a node n consists of the successors of all nodes dominated by n that are not strictly dominated by n.

  $$DF(n) =\{m \mid \exists p \in preds(m) \text{ where } n \underline{\gg} p \wedge !(n \gg m)\}$$

- DF(n) is the set of nodes where a join point for a definition of a variable in n can occur

Computing Dominance Frontiers

$DF(n) = DF_{local}(n) \cup (\bigcup_{c \in child(n)} DF_{up}(c))$

$DF_{local}(n) = \{m \mid m \in succ(n) \wedge !(n >> m)\}$

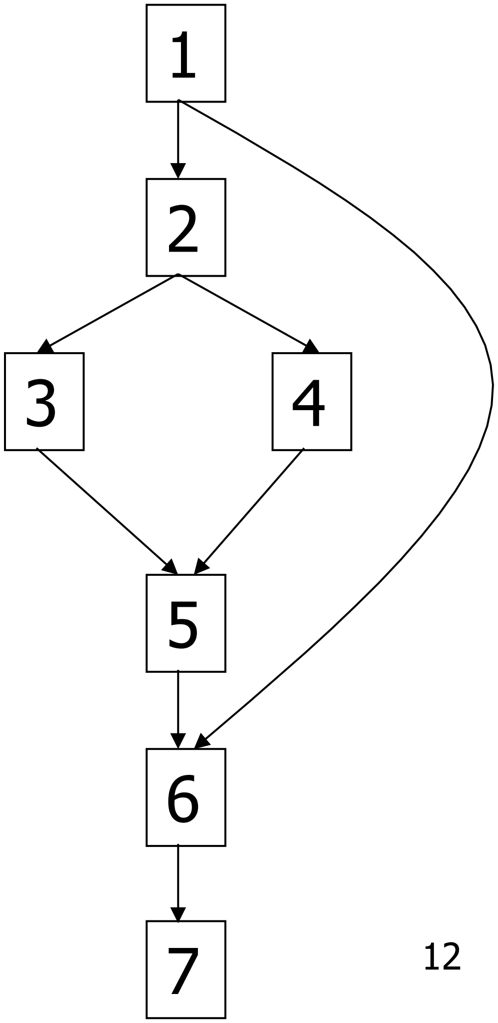
$DF_{up}(c) = \{m \mid m \in DF(c) \wedge !(idom(c) >> m)\}$

- $DF_{local}(n)$ is the dominance frontier of n involving only the successors of n.
- $DF_{up}(c)$ propagates DF_{local} information up the dominator tree. Includes everything in the dominance frontier of the children of n that n does not dominate itself, excluding n.

Computing Dominance Frontiers

```
for each $n \in$ DT in postorder {
    DF(n) = $\emptyset$
    for each $c \in$ child(n) //DT
      for each $m \in$ DF(c)
        if !(n >> m)
          DF(n) $\cup$= {m}
    for each $m \in$ succ(n)
      if !(n >> m)
        DF(n) $\cup$= {m}
}
```

- Compute the dominance frontier for the example



Placement of ϕ-nodes

- Let S_v be the set of all blocks with assignments to v plus the ENTRY node.

  $$DF(S_v) = \cup_{n \in S_v} DF(n)$$

  This is the set of all possible join points for assignments to v.
- If we place ϕ-nodes in each $b \in DF(S_v)$ will this be correct?
  - Does S_v contain all blocks in the dominance frontier of all blocks with definitions of v?

Iterated Dominance Frontier

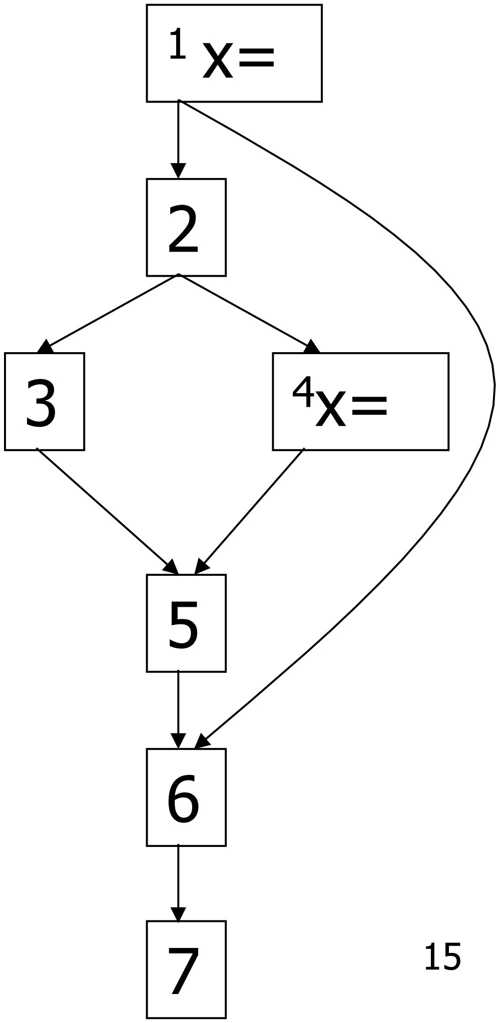
- $DF^{+}(S_v)$ is the iterated dominance frontier for the set of definitions S_v
  - New blocks are potentially added for each ϕ-node insertion.
- Computing $DF^{+}(S_v)$

$$DF_1(S_v) = DF(S_v)$$
$$DF_{i+1}(S_v) = DF(S_v \cup DF_i(S_v))$$
$$DF^{+}(S_v) = \cup_{i=1,\infty} DF_i(S_v)$$

Iterated Dominance Frontier

```
Work = $\emptyset$
DF$^+$(S$_v$) = $\emptyset$
for each b $\in$ S$_v$ {
      Work $\cup$= {b}
}
while Work $\neq$ $\emptyset$ {
     b = Work.Remove()
     for each c $\in$ DF(b)
       if c $\notin$ DF$^+$(S$_v$) {
         DF$^+$(S$_v$) $\cup$= {c}
         Work $\cup$= {c}
       }
}
```

- Compute the iterated dominance frontier for the example



Inserting ϕ-nodes

```
Perform live-variable analysis
for each T ∈ Variables
    if T ∈ Globals {
      S = {b| b has a def of T}
           ∪ {Entry}
      Compute DF+(S)
      for each b ∈ DF+(S)
        if T ∈ b.LiveIn {
          n = | pred(b) |
          insert T=ϕ(T_1,…,T_n) in b
        }
    }
```

- Insert ϕ-nodes for previous example.
- leave parameters to ϕ-nodes named by path
  - renaming will get the correct names

Renaming Temporaries

- Need to replace uses with new names
  - walk the dominator tree
  - replace uses dominated by a definition

```
for each T ∈ Variables
   NameStack(T) = ∅
Rename(ENTRY)
```

Renaming Algorithm

```
Rename(b) {
    for each $I \in \Phi(b)$ of the form $T_0 = \phi(T_1,\ldots,T_n)$ {
      push NewName() on NameStack($T_0$)
      Definition(Top(NameStack($T_0$)) = I
    }
    for each $I \in b$ in order {
      for each $T \in$ Operand(I) {
        replace T by Top(NameStack(T))
        add I to Uses(Top(NameStack(T)))
      }
      T = Target(I)
      push NewName() on NameStack(T)
      Definition(Top(NameStack(T)) = I
    }
```

Renaming Algorithm Contd.

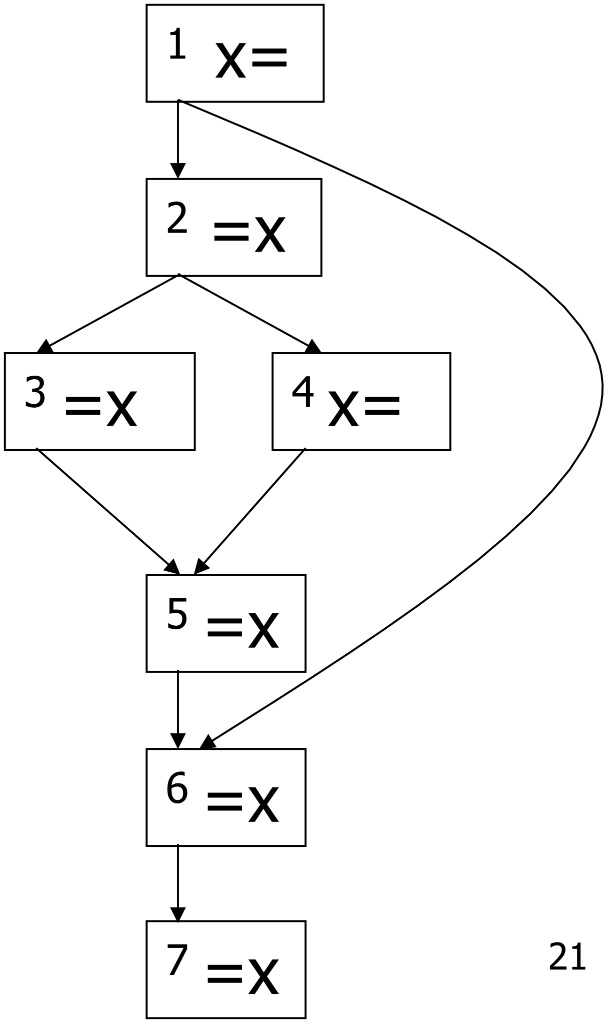
```
for each $s \in$ succ(b) { // successor in CFG
  j = WhichPredecessor(s,b)
  for each $I \in \Phi(s)$ of the form $T_0 = \phi(T_1,\ldots,T_n)$ {
    replace $T_j$ by Top(NameStack($T_j$))
    add I to Uses(Top(NameStack($T_j$)))
  }
}
for each $c \in$ Children(b) // in dominator tree
  Rename(c)
```

Renaming Algorithm Contd.

```
   for each I $\in$ b in reverse order {
     T = Target(I)
     replace T by Pop(NameStack(T))
   }
   for each I $\in \Phi$(b) of the form $T_0 = \phi(T_1,\ldots,T_n)$ {
     replace $T_0$ by Pop(NameStack($T_0$))
}
```

Example

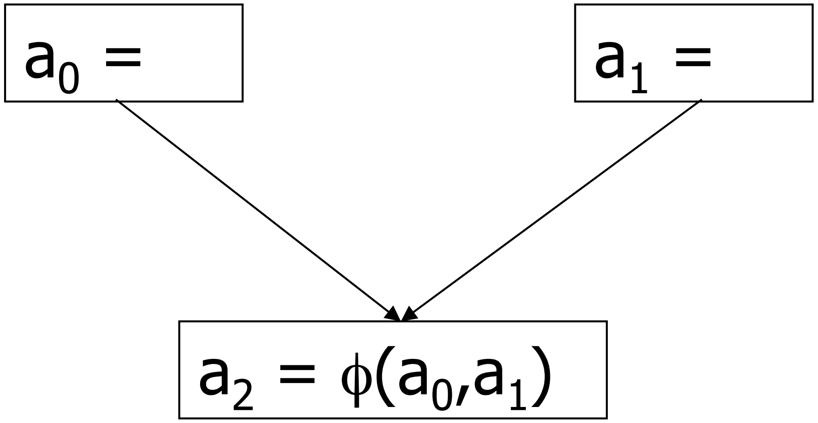
- Convert the code to the right to SSA

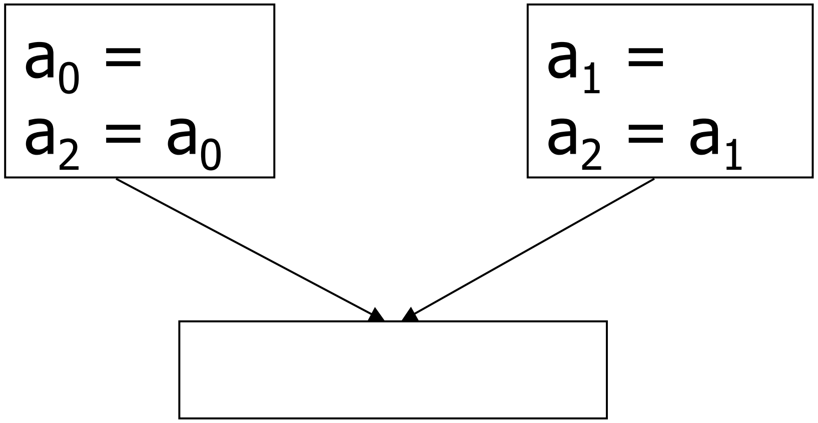


SSA to Normal Form

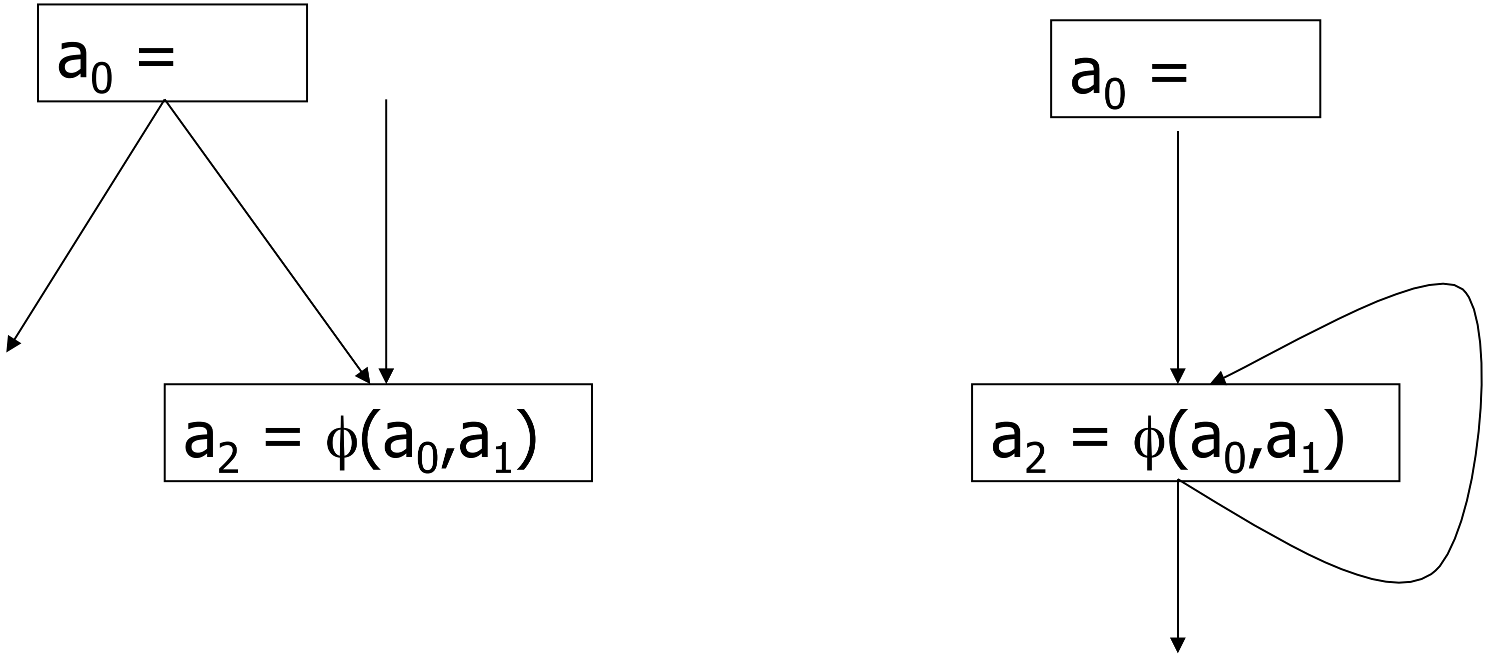
- ϕ-nodes require copies from operands to l-value for each operand

- Becomes





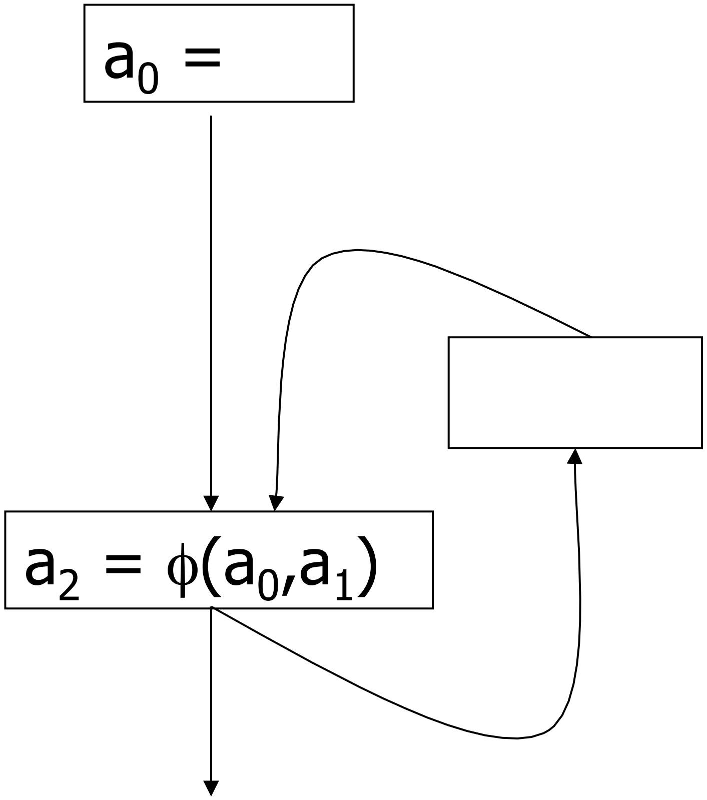
Problems with Direct Translation



- Cannot move a copy into predecessor
- Cannot put copy at beginning of block

Critical Edges

- An edge where the tail of the edge has more than one predecessor and the head of the edge has more than one successor is called the critical edge.
- The solution is to insert a basic block on all critical edges so that the CFG has none.



Abnormal Edges

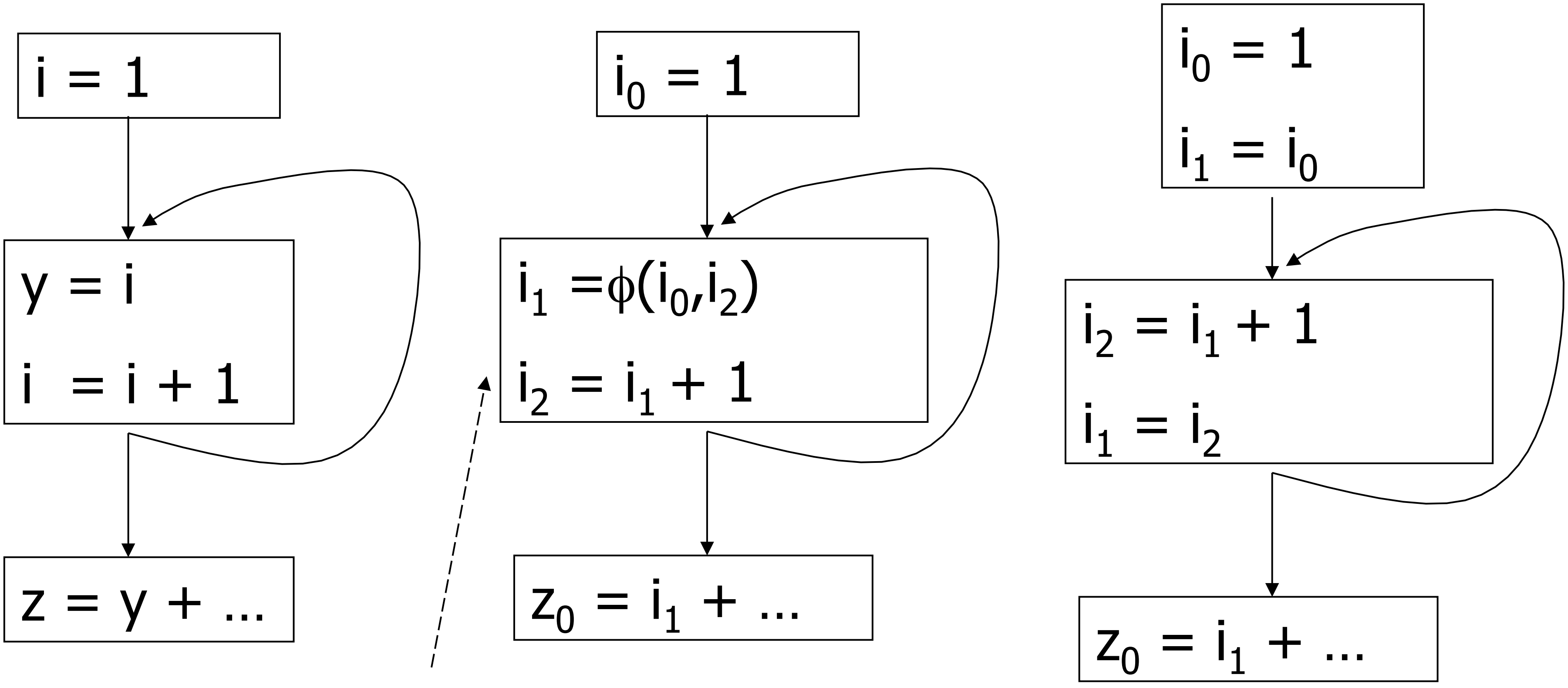
- Edges where the head is not definite (known branch target) are called abnormal edges.

```
switch(a) {
case 1:
    // no break statement
case 2:
}
```

```
    iLD       a, r1
    iMULI     8, r1, r2
    iLDA      br_table, r1
    BR        r2(r1)
    …
L1: nop
    …
L2: nop
```

Must ensure that no blocks will need to be inserted on an abnormal critical edge

The Lost-Copy Problem



$i = 1$

$y = i$

$i \ = i + 1$

$z = y + \ldots$

$i_0 = 1$

$i_1 = \phi(i_0, i_2)$

$i_2 = i_1 + 1$

$z_0 = i_1 + \ldots$

$i_0 = 1$

$i_1 = i_0$

$i_2 = i_1 + 1$

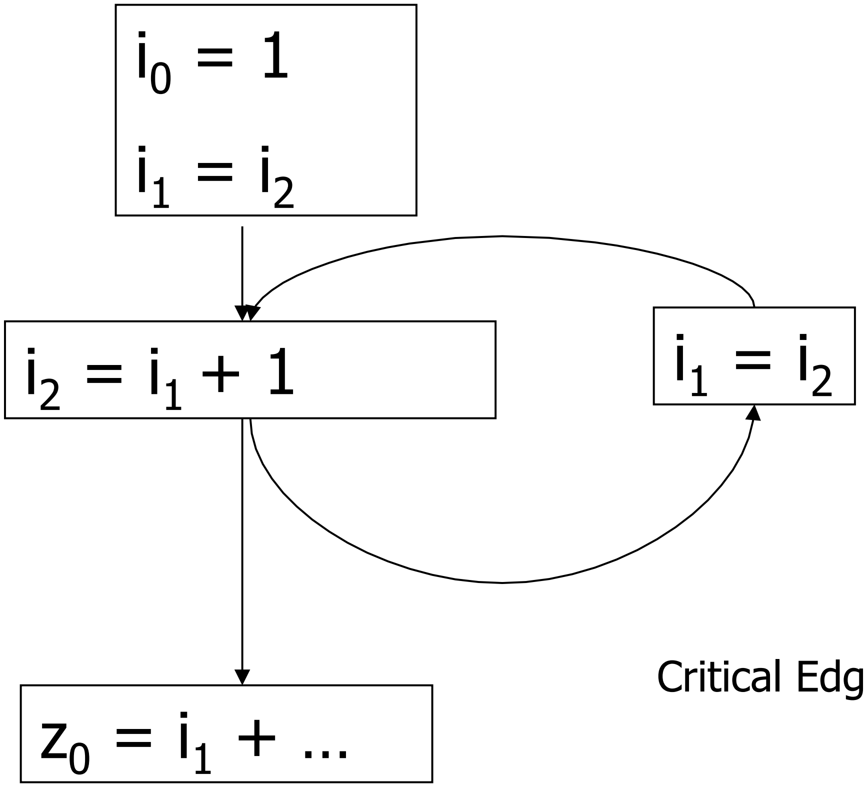
$i_1 = i_2$

$z_0 = i_1 + \ldots$

Copy folding
applied

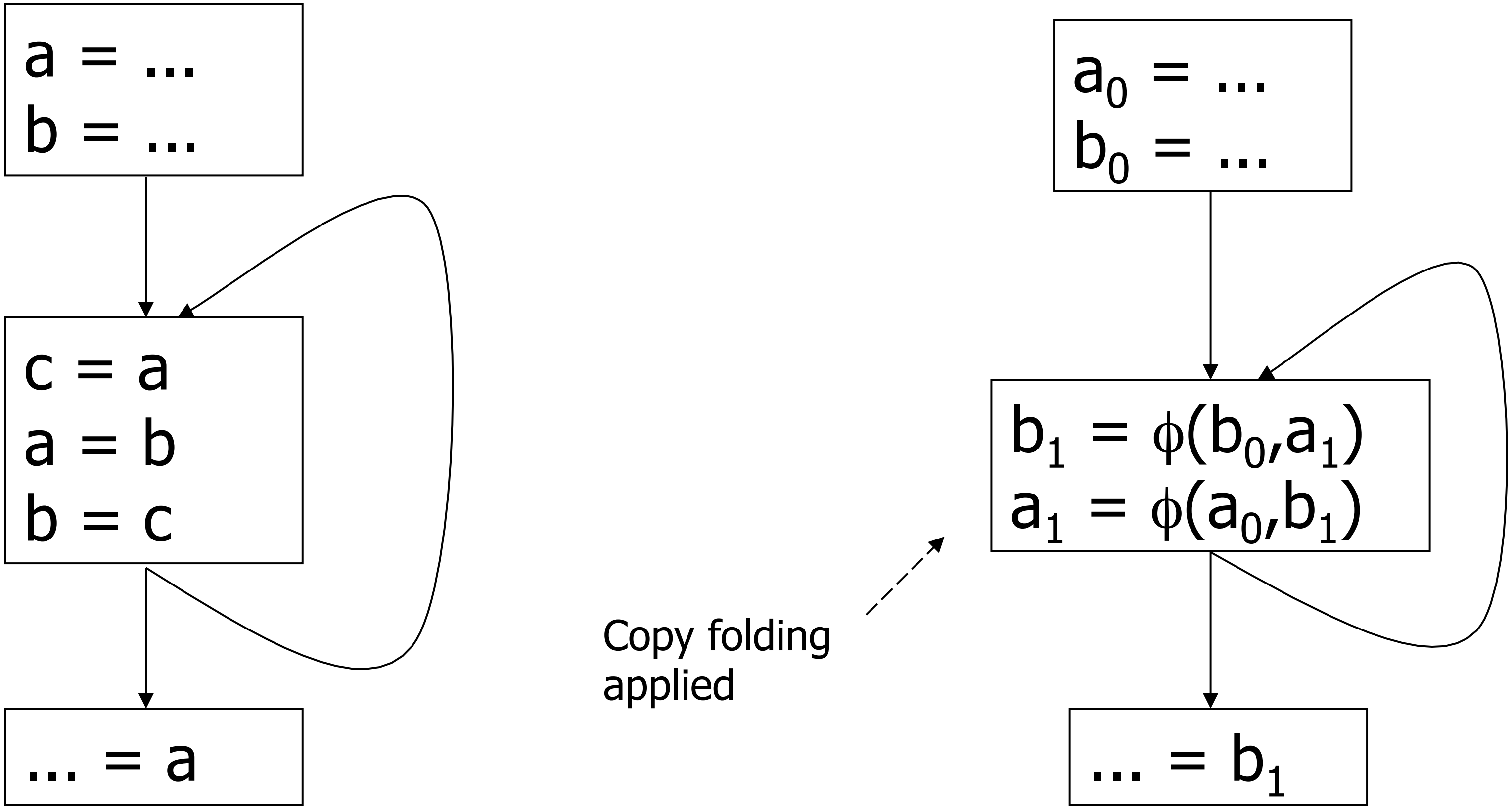
Incorrect insertion

The Lost-Copy Problem



Critical Edge Split

The Swap Problem



a = …
b = …

c = a
a = b
b = c

… = a

a_0 = …
b_0 = …

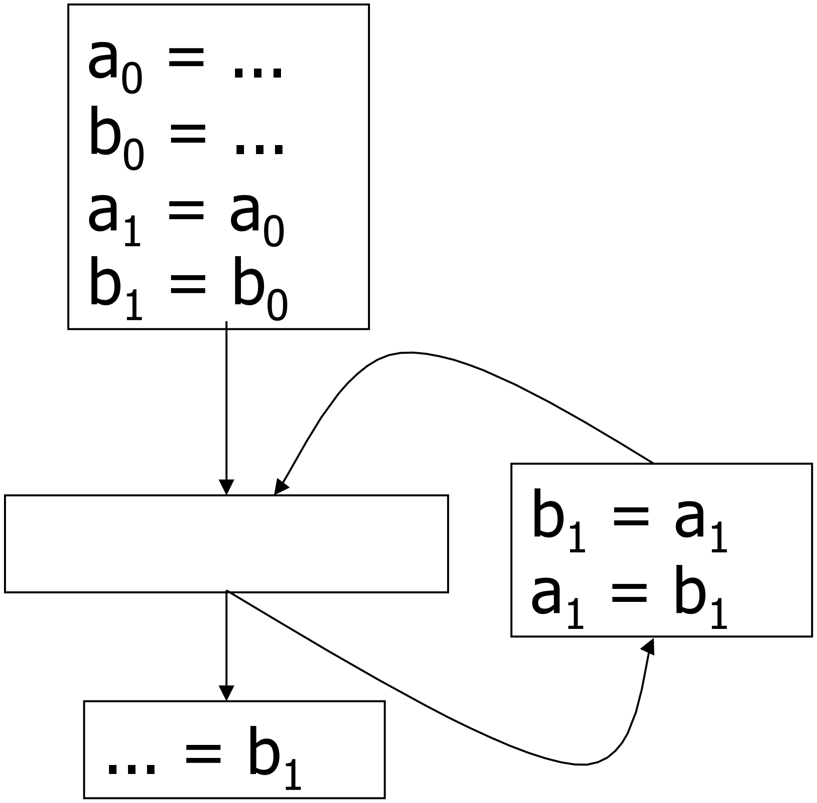
$b_1 = \phi(b_0,a_1)$
$a_1 = \phi(a_0,b_1)$

… = b_1

Copy folding applied

The Swap Problem

- In a block b, all members of $\Phi(b)$ are executed simultaneously
- Direct translation of the previous code results in incorrect code.

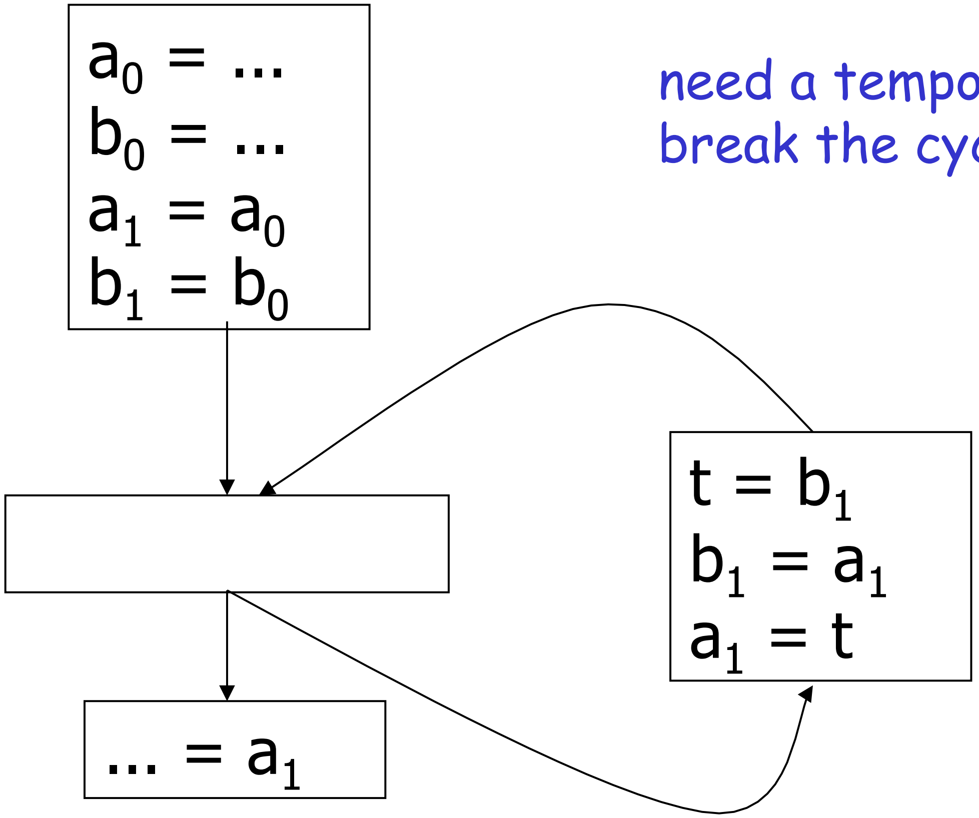


$a_0 = \ldots$
$b_0 = \ldots$
$a_1 = a_0$
$b_1 = b_0$

$b_1 = a_1$
$a_1 = b_1$

$\ldots = b_1$

Correct Translation



a_0 = …
b_0 = …
$a_1 = a_0$
$b_1 = b_0$

$t = b_1$
$b_1 = a_1$
$a_1 = t$

… = a_1

need a temporary to break the cycle

Translating to Normal Form

- Given a CFG in SSA form and a partition $P=\{P_1,\ldots P_n\}$ of the set of all variables, rewrite the CFG in normal form so that any two temporaries T_1 and T_2 in P_i are given the same temporary name and ϕ-nodes are replaced by equivalent copy operations. The partition must ensure that
  - In each block b, if two targets of ϕ-nodes are equivalent, then the corresponding arguments must be equivalent.
  - For each abnormal critical edge (c,b) if $T_0=\phi(T_1,\ldots,T_i,\ldots.T_n)$ is a ϕ-node in b and c is the i^{th} predecessor of b, then T_0 and T_i must be equivalent (no copies on abnormal critical edges).
- Each P_i has a single unique name
- Can use global value numbering to compute partition

Renaming ϕ-nodes

- Since all $i \in \Phi(b)$ are executed simultaneously, they need to be topologically sorted so that all uses of a variable T_i are generated before the definition.
- Since there may be cycles, these need to be handled separately
  - find cycles
  - break cycles with an additional temporary

Cycles within ϕ-nodes

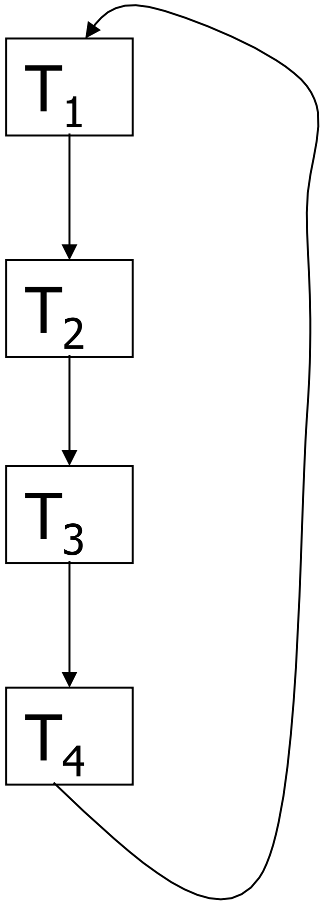
- A graph R(b) such that the nodes are the elements of P and there is an edge from FIND(T_k) to FIND (T_l) if there are temporaries T_k and T_l such that $T_k=\phi(\ldots,T_l,\ldots) \in \Phi(b)$.
- Use Tarjan's SCC algorithm to find cycles in R(b).

Cycles within ϕ-nodes

- For each SCC do the following
  1. Enumerate the cycle in some topological order such that the first node is a successor of the last.
  2. Generate one extra temporary, T.
  3. Generate an instruction to copy the temporary representing the first node into T.
  4. Translate all of the other nodes except the last one normally.
  5. Generate an instruction to copy T into the temporary corresponding to the final node.

Example

$T_1 = \phi(\ldots, T_2, \ldots)$
$T_2 = \phi(\ldots, T_3, \ldots)$
$T_3 = \phi(\ldots, T_4, \ldots)$
$T_4 = \phi(\ldots, T_1, \ldots)$



Algorithm

```
foreach b $\in$ G {
    foreach i $\in$ b {
      foreach T $\in$ Operands(i)
          replace T by FIND(T)
      foreach T $\in$ Targets(i)
          replace T by FIND(T)
      if i = (T = T)
          delete i from b
    }
    foreach c $\in$ pred(b)
      call eliminate-$\phi$(c,b,whichpred(c,b))
}
foreach b $\in$ G
    remove $\phi$-nodes from b
```

Algorithm

```
procedure eliminate-$\phi$(c,b,i)
    call eliminateBuild(b,i)
    if nodeSet $\neq \emptyset$ {
      Visited = Stack = $\emptyset$
      foreach T $\in$ nodeSet
            if T $\notin$ Visited
              call elimForward(T)
      Visited = $\emptyset$
      while Stack $\neq \emptyset$ {
            pop T from Stack
            if T $\notin$ Visited
              call elimCreate(T)
      }
    }
end eliminate-$\phi$
```

```
procedure elimForward(T)
    add T to Visited
    foreach S $\in$ elimSucc(T)
      if S $\notin$Visited
            elimForward(S)
    push T onto Stack
end elimForward
```

Algorithm

```
procedure eliminateBuild(b,i)
    nodeSet = $\emptyset$
    foreach $T_0=\phi(\ldots,T_i,\ldots)\in\Phi(b)$
      $x_0$ = FIND($T_0$)
      $x_1$ = FIND($T_i$)
      if $x_0 \neq x_1$ {
            call elimName($x_0$)
            call elimName($x_1$)
            add $x_0$ to elimPred($x_1$)
            add $x_1$ to elimSucc($x_0$)
    }
end eliminateBuild
```

```
procedure elimName(T)
    if $T \notin$ nodeSet {
      add T to nodeSet
      elimSucc(T) = $\emptyset$
      elimPred(T) = $\emptyset$
    }
end elimName
```

Algorithm

```
procedure elimCreate(T)
    if elimUnvisitPred(T) {
      create new temp U
      append “U=T” to C
      foreach $p \in$ elimPred(T)
        if $p \notin$ Visited {
            call elimBack(p)
            append “P=U” to C
        }
    }
    else if elimSucc(T) $\neq \emptyset$ {
      add T to Visited
      take S from elimSucc(T)
      append “T=S” to C
    }
end elimCreate
```

```
function elimUnvisitPred(T)
    foreach $p \in$ elimPred(T)
      if $p \notin$ Visited
       return true
    return false
end elimUnvisitPred
```

```
procedure elimBack(T)
    add T to Visited
    foreach $p \in$ elimPred(T)
      if $p \notin$ Visited {
        call elimBack(p)
        append “P=T” to C
      }
end elimBack
```

Example

$F = \phi(\ldots, B, \ldots)$

$C = \phi(\ldots, D, \ldots)$

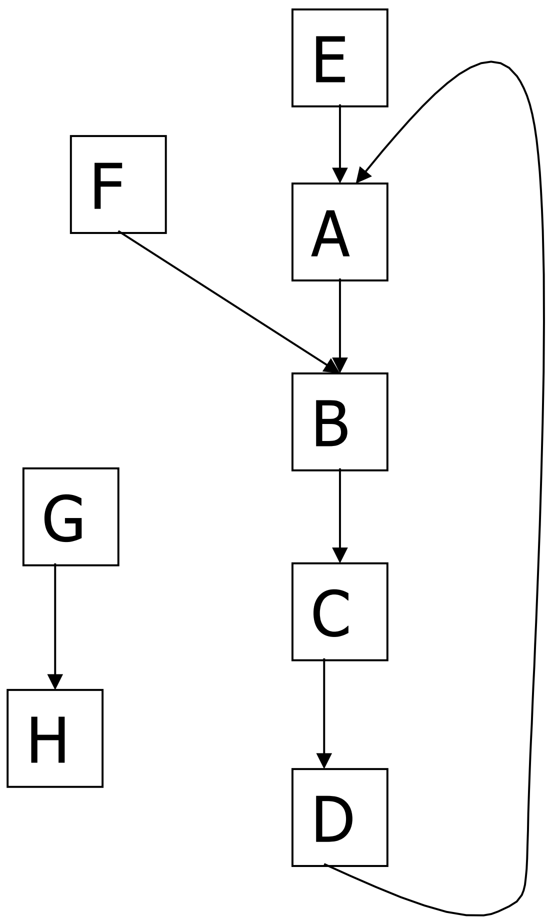
$E = \phi(\ldots, A, \ldots)$

$G = \phi(\ldots, H, \ldots)$

$B = \phi(\ldots, C, \ldots)$

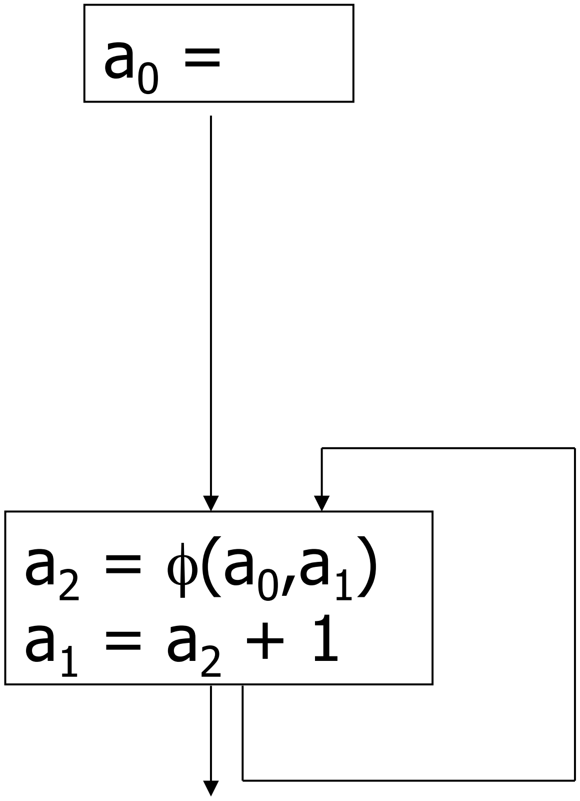
$D = \phi(\ldots, A, \ldots)$

$A = \phi(\ldots, B, \ldots)$



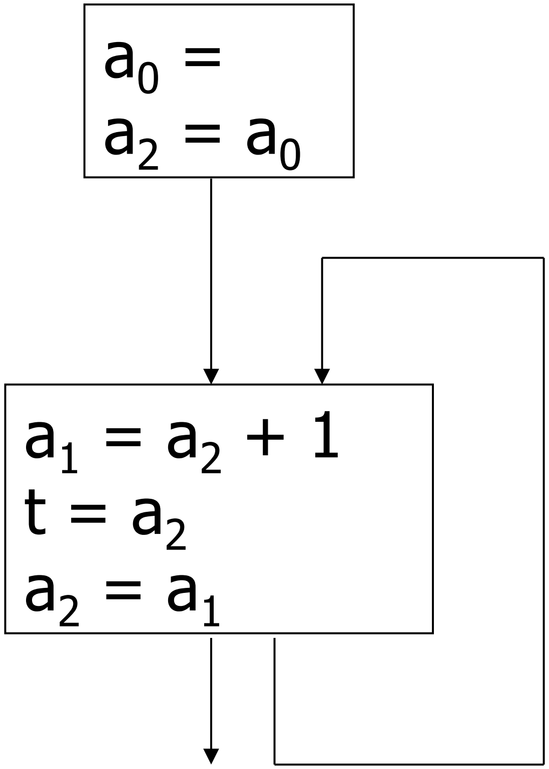
Critical Edges Re-visited

- In the CFG to the right, if a_2 is not used outside the block, the new basic block is unnecessary.
- Since inserting a block on a back edge puts a jump in loop, splitting the critical edge is not advisable.



Critical Edges Re-visited

- If a_2 is used outside the block, add a copy to a temporary t and replace the uses of a_2 outside the block with t


$a_0 =$
$a_2 = a_0$
$a_1 = a_2 + 1$
$t = a_2$
$a_2 = a_1$