# EECS 583 – Class 7
# Static Single Assignment Form

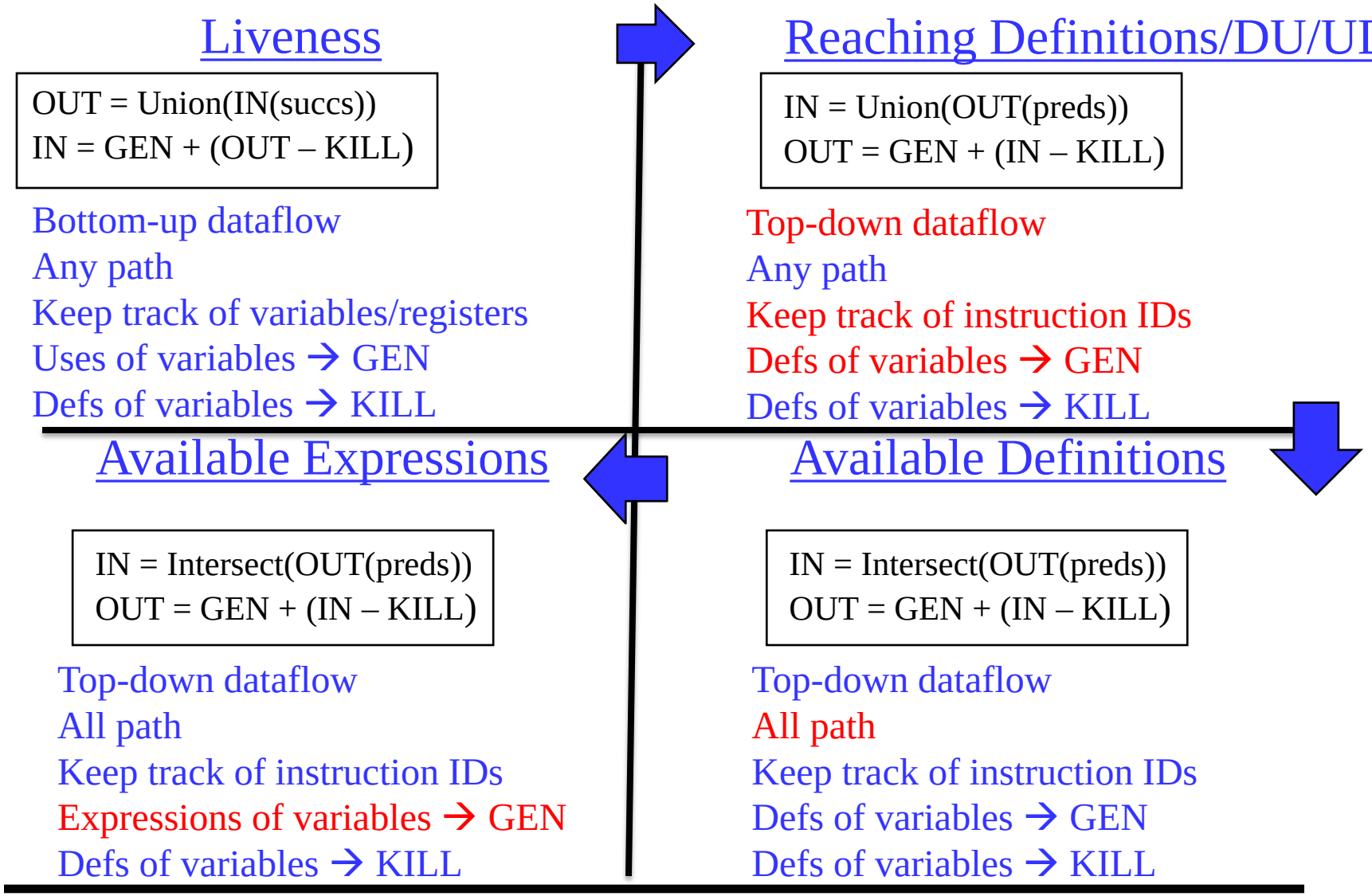
*University of Michigan*

*September 28, 2011*

# Reading Material

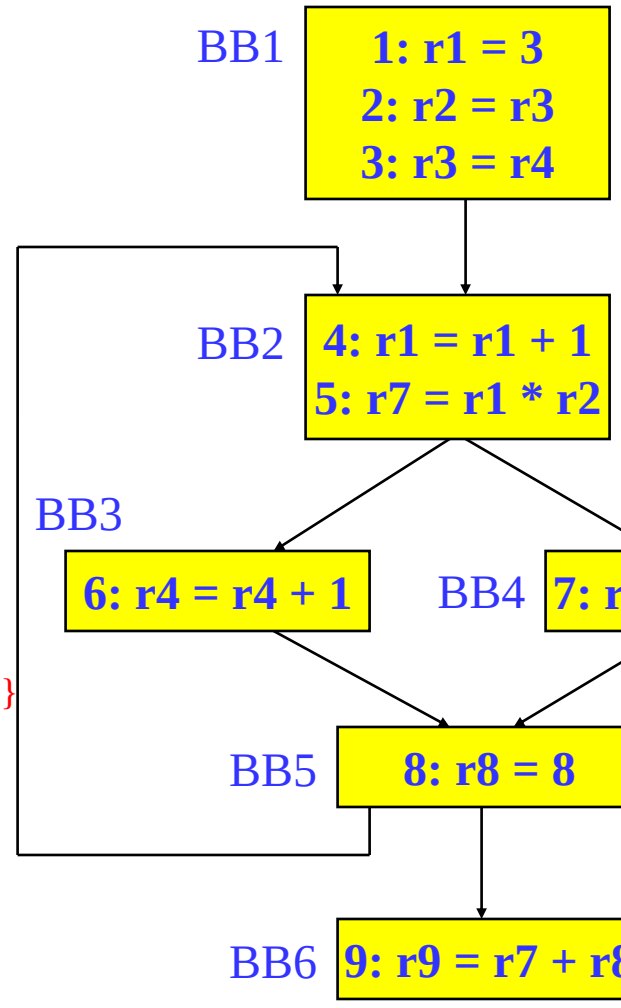
- Today's class
  - "Practical Improvements to the Construction and Destruction of Static Single Assignment Form," P. Briggs, K. Cooper, T. Harvey, and L. Simpson, *Software--Practice and Experience*, 28(8), July 1998, pp. 859-891.
- Next class – Optimization
  - *Compilers: Principles, Techniques, and Tools*, A. Aho, R. Sethi, and J. Ullman, Addison-Wesley, 1988, 9.9, 10.2, 10.3, 10.7

# Last Class in 1 Slide



# From Last Time: Class Problem - Rdefs

Compute reaching defs
- Calculate GEN/KILL for each BB
- Calculate IN/OUT for each BB




IN = {}
Gen = {1,2,3}
Kill = {4}
OUT = {1,2,3}

Remember, initial value for the OUT of BB5 is Gen(BB5) = 8.

IN = {1,2,3,8} → {1,2,3,4,5,6,7,8}
Gen = {4,5}
Kill = {1}
OUT = {2,3,4,5,8} → {2,3,4,5,6,7,8}

IN = {2,3,4,5,8} → {2,3,4,5,6,7,8}
Gen = {6}
Kill = {7}
OUT = {2,3,4,5,6,8} → {2,3,4,5,6,8}

IN = {2,3,4,5,8} → {2,3,4,5,6,7,8}
Gen = {7}
Kill = {6}
OUT = {2,3,4,5,7,8} → {2,3,4,5,7,8}

IN = {2,3,4,5,6,7,8}
Gen = {8}
Kill = {}
OUT = {2,3,4,5,6,7,8}

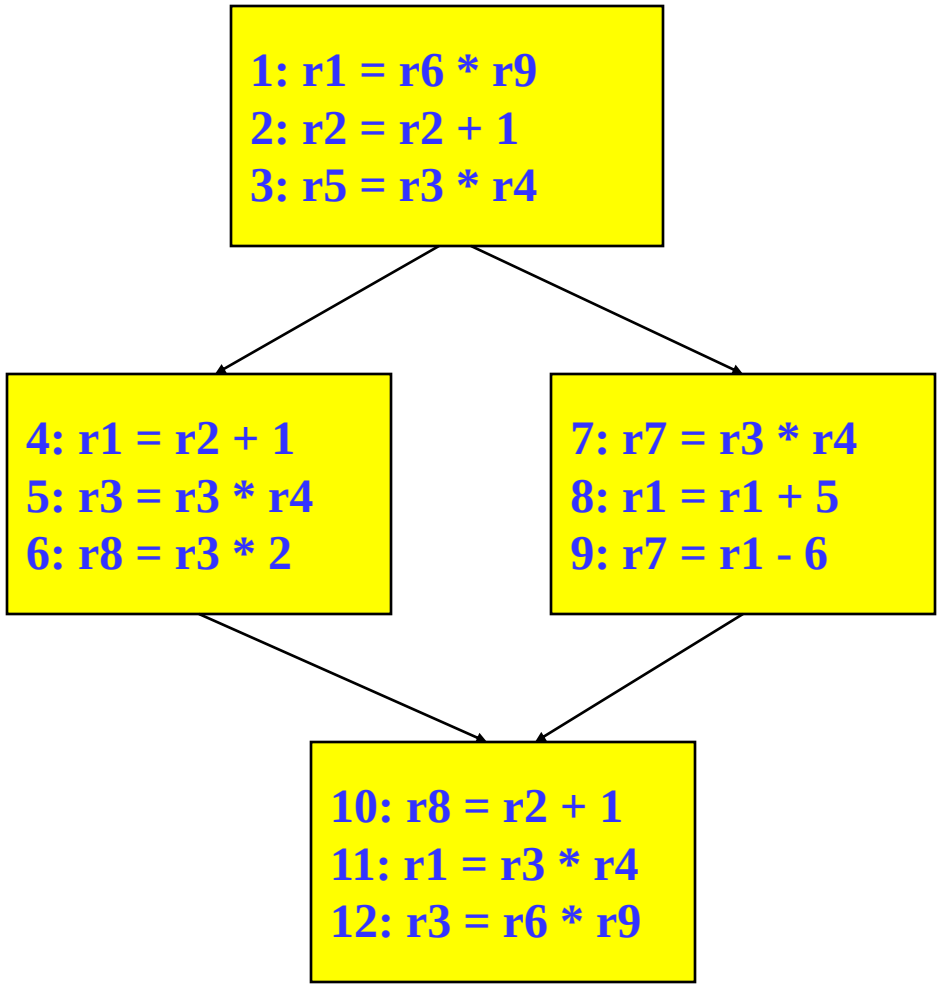
IN = {2,3,4,5,6,7,8}
Gen = {9}
Kill = {}
OUT = {2,3,4,5,6,7,8,9}

# From Last Time: Computation of Aexpr GEN/KILL Sets

We can also formulate the GEN/KILL slightly differently so you do not need to break up instructions like "r2 = r2 + 1".

```
for each basic block in the procedure, X, do
   GEN(X) = 0
   KILL(X) = 0
   for each operation in sequential order in X, op, do
      K = 0
      for each destination operand of op, dest, do
         K += {all ops which use dest}
      endfor
      if (op not in K)
            G = op
      else
            G = 0
      GEN(X) = G + (GEN(X) – K)
      KILL(X) = K + (KILL(X) – G)
   endfor
endfor
```

# Class Problem - Aexprs Calculation

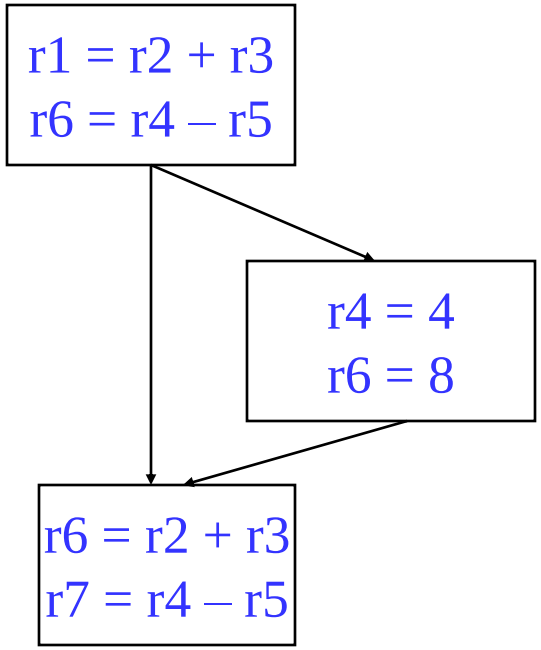


# Some Things to Think About

- Liveness and rdefs are basically the same thing
  - All dataflow is basically the same with a few parameters
    - Meaning of gen/kill – src vs dest, variable vs operation
    - Backward / Forward
    - All paths / some paths (must/may)
- Dataflow can be slow
  - How to implement it efficiently?
    - Forward analysis – DFS order
    - Backward analysis – PostDFS order
  - How to represent the info? → Bitvectors
- Predicates
  - Throw a monkey wrench into this stuff
  - So, how are predicates handled?
    - See “Analysis techniques for predicated code,” R. Johnson and M. Schlansker, MICRO 1996.

# Static Single Assignment (SSA) Form

- Difficulty with optimization
  - Multiple definitions of the same register
  - Which definition reaches
  - Is expression available?

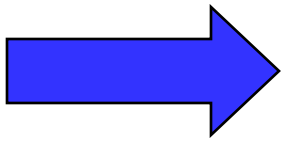



- Static single assignment
  - Each assignment to a variable is given a unique name
  - All of the uses reached by that assignment are renamed
  - DU chains become obvious based on the register name!

# Converting to SSA Form

- Trivial for straight line code

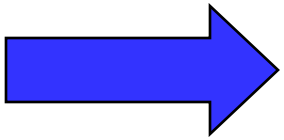
```
x = -1
y = x
x = 5
z = x
```



```
x0 = -1
y = x0
x1 = 5
z = x1
```

- More complex with control flow – Must use Phi nodes

```
if ( ... )
    x = -1
else
    x = 5
y = x
```

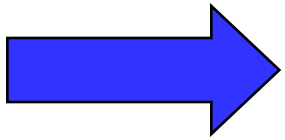


```
if ( ... )
    x0 = -1
else
    x1 = 5
x2 = Phi(x0,x1)
y = x2
```

# Converting to SSA Form (2)

- What about loops?
  - No problem!, use Phi nodes again

```
i = 0
do {
   i = i + 1
}
while (i < 50)
```



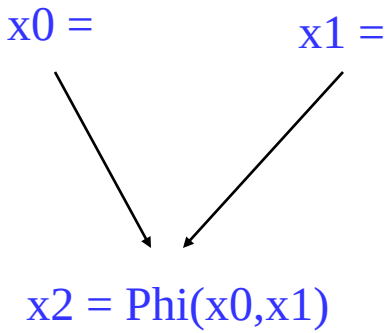
```
i0 = 0
do {
   i1 = Phi(i0, i2)
   i2 = i1 + 1
}
while (i2 < 50)
```

# SSA Plusses and Minuses

- Advantages of SSA
  - Explicit DU chains – Trivial to figure out what defs reach a use
    - Each use has exactly 1 definition!!!
  - Explicit merging of values
  - Makes optimizations easier
- Disadvantages
  - When transform the code, must either recompute (slow) or incrementally update (tedious)

# Phi Nodes (aka Phi Functions)

- Special kind of copy that selects one of its inputs
- Choice of input is governed by the CFG edge along which control flow reached the Phi node



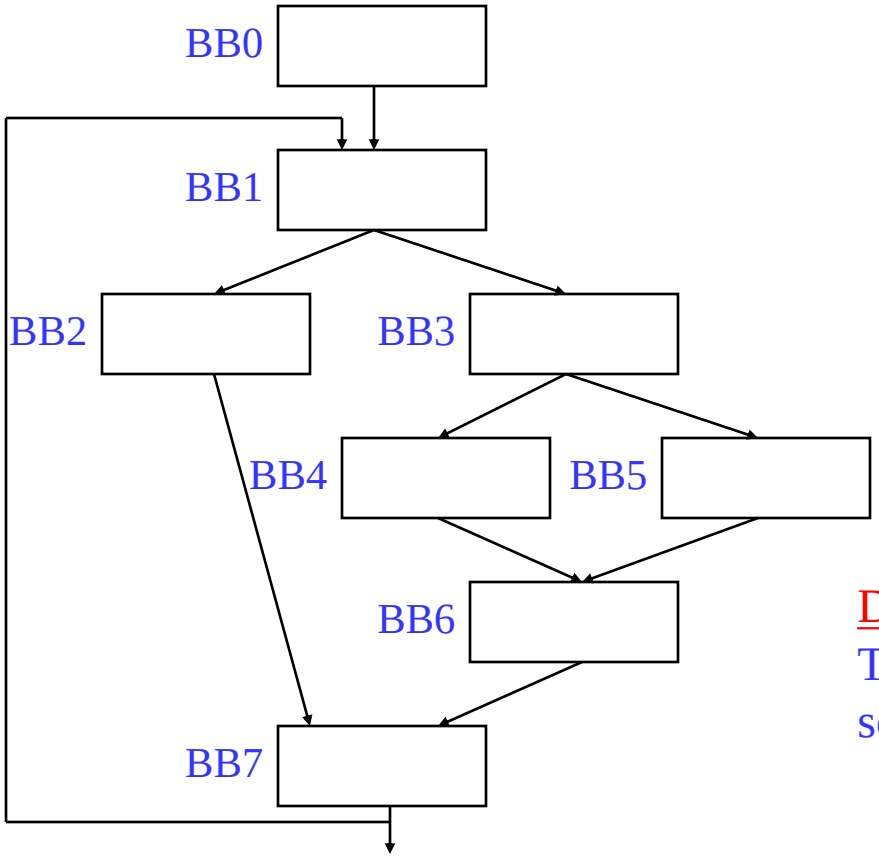

- Phi nodes are required when 2 non-null paths X→Z and Y→Z converge at node Z, and nodes X and Y contain assignments to V

# SSA Construction

- High-level algorithm
  1. Insert Phi nodes
  2. Rename variables
- A dumb algorithm
  - Insert Phi functions at every join for every variable
  - Solve reaching definitions
  - Rename each use to the def that reaches it (will be unique)
- Problems with the dumb algorithm
  - Too many Phi functions (precision)
  - Too many Phi functions (space)
  - Too many Phi functions (time)

# Need Better Phi Node Insertion Algorithm

- A definition at n forces a Phi node at m iff n not in DOM(m), but n in DOM(p) for some predecessors p of m




def in BB4 forces Phi in BB6
def in BB6 forces Phi in BB7
def in BB7 forces Phi in BB1

Phi is placed in the block that
is just outside the dominated region
of the definition BB

Dominance frontier

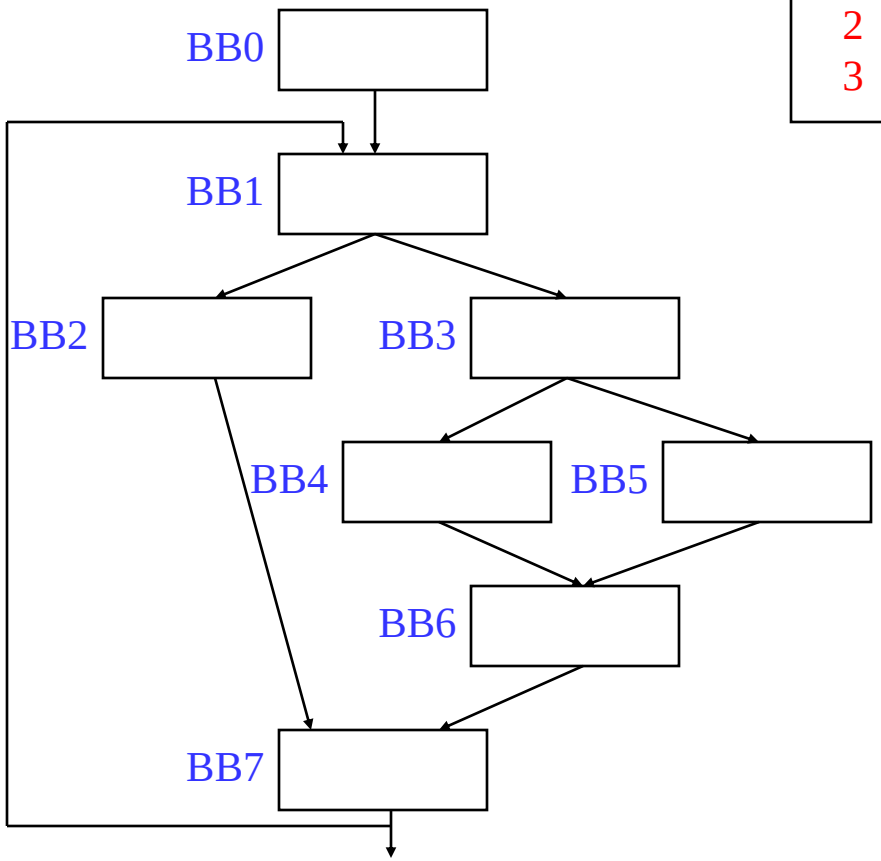
The dominance frontier of node X is the set of nodes Y such that

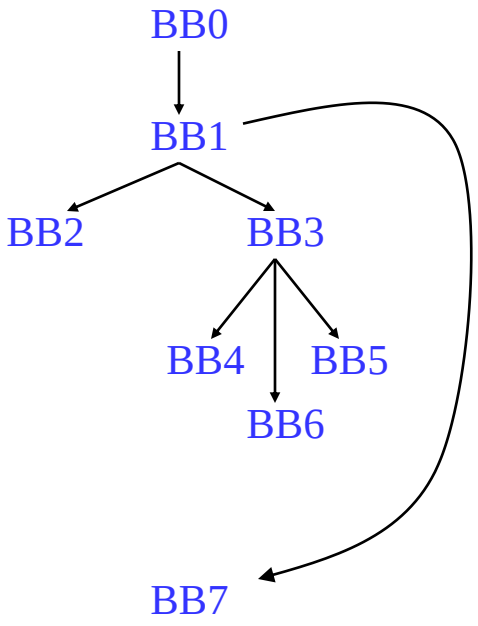
* X dominates a predecessor of Y, but
* X does not strictly dominate Y

# Dominator Tree

First BB is the root node, each node dominates all of its descendants

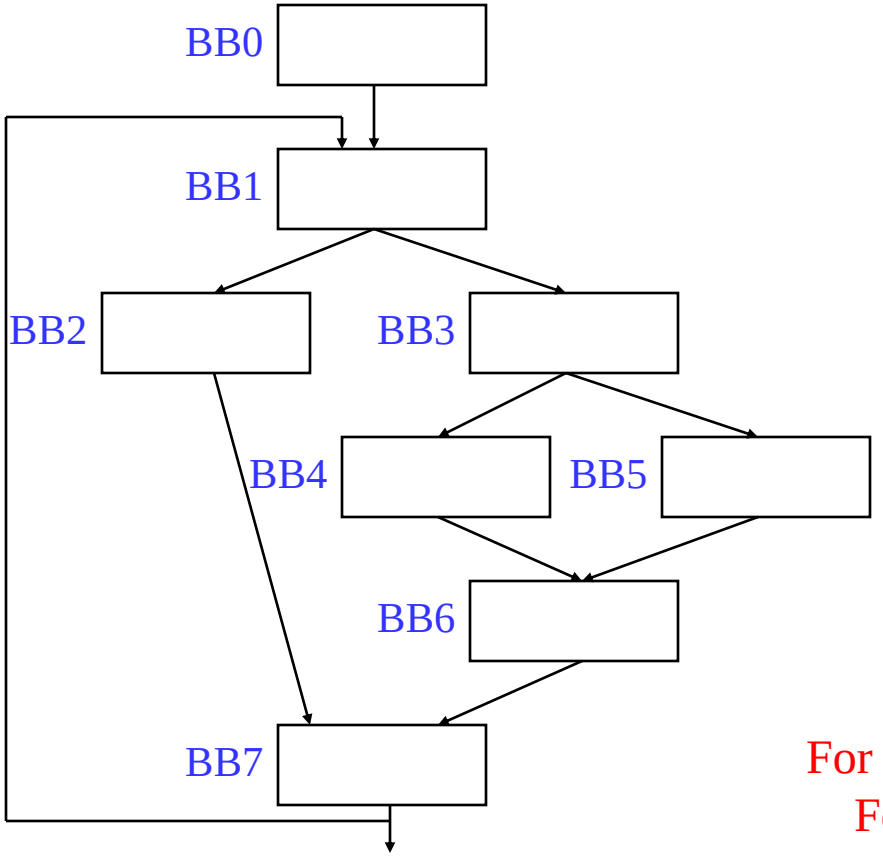
| BB | DOM | BB | DOM |
|---|---|---|---|
| 0 | 0 | 4 | 0,1,3,4 |
| 1 | 0,1 | 5 | 0,1,3,5 |
| 2 | 0,1,2 | 6 | 0,1,3,6 |
| 3 | 0,1,3 | 7 | 0,1,7 |

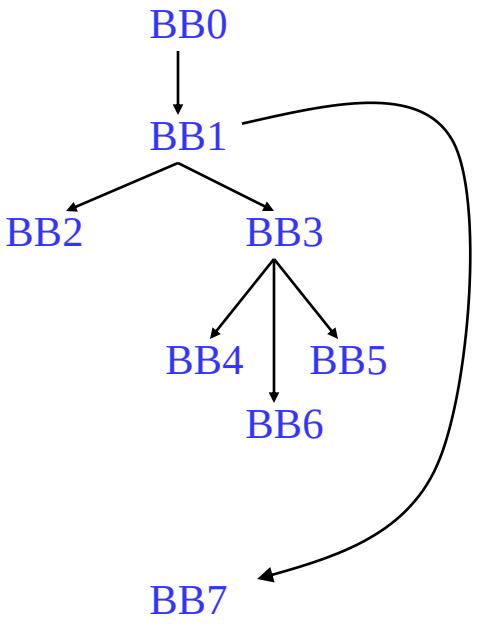



**Dom tree**

# Computing Dominance Frontiers



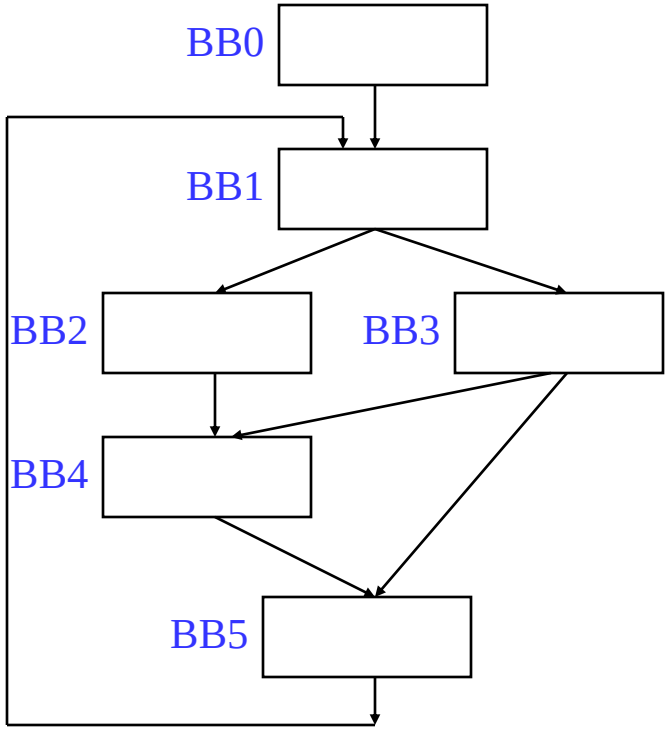




| BB | DF |
|---|---|
| 0 | - |
| 1 | - |
| 2 | 7 |
| 3 | 7 |
| 4 | 6 |
| 5 | 6 |
| 6 | 7 |
| 7 | 1 |

For each join point X in the CFG
  For each predecessor, Y, of X in the CFG
    Run up to the IDOM(X) in the dominator tree, adding X to DF(N) for each N between Y and IDOM(X)

# Class Problem

Draw the dominator tree, calculate the dominance frontier for each BB

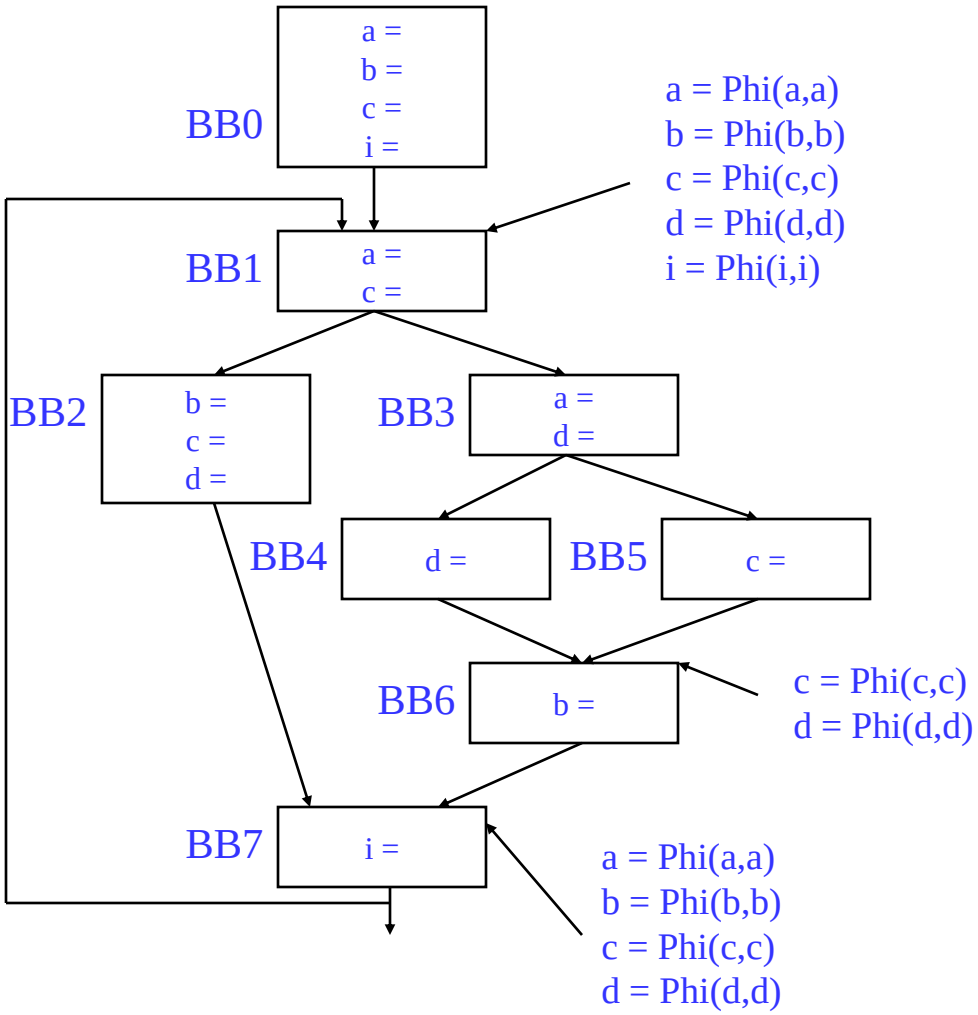


# Phi Node Insertion Algorithm

- Compute dominance frontiers
- Find global names (aka virtual registers)
  - Global if name live on entry to some block
  - For each name, build a list of blocks that define it
- Insert Phi nodes
  - For each global name n
    - For each BB b in which n is defined
      - For each BB d in b's dominance frontier
        - Insert a Phi node for n in d
        - Add d to n's list of defining BBs

# Phi Node Insertion - Example

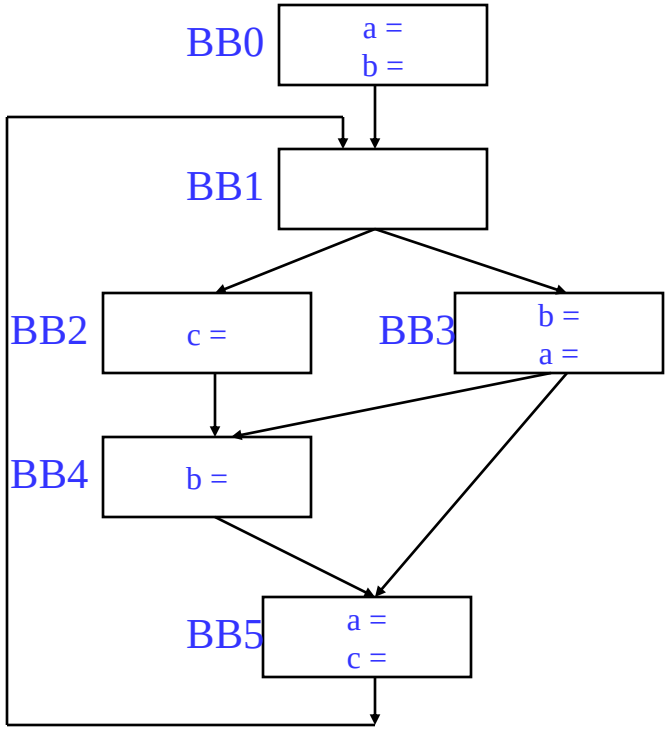
| BB | DF |
|---|---|
| 0 | - |
| 1 | - |
| 2 | 7 |
| 3 | 7 |
| 4 | 6 |
| 5 | 6 |
| 6 | 7 |
| 7 | 1 |




a is defined in 0,1,3
need Phi in 7
then a is defined in 7
need Phi in 1
b is defined in 0, 2, 6
need Phi in 7
then b is defined in 7
need Phi in 1
c is defined in 0,1,2,5
need Phi in 6,7
then c is defined in 7
need Phi in 1
d is defined in 2,3,4
need Phi in 6,7
then d is defined in 7
need Phi in 1
i is defined in BB7
need Phi in BB1

# Class Problem

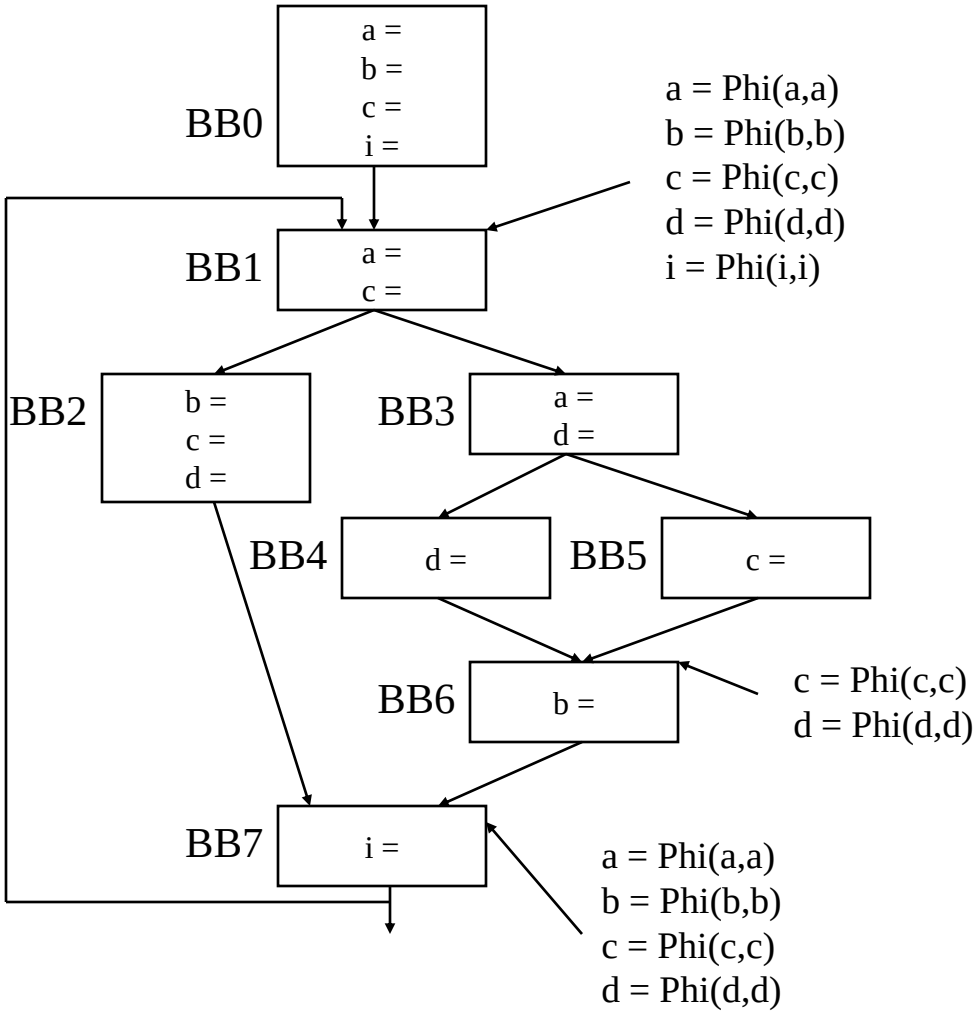
Insert the Phi nodes



# SSA Step 2 – Renaming Variables

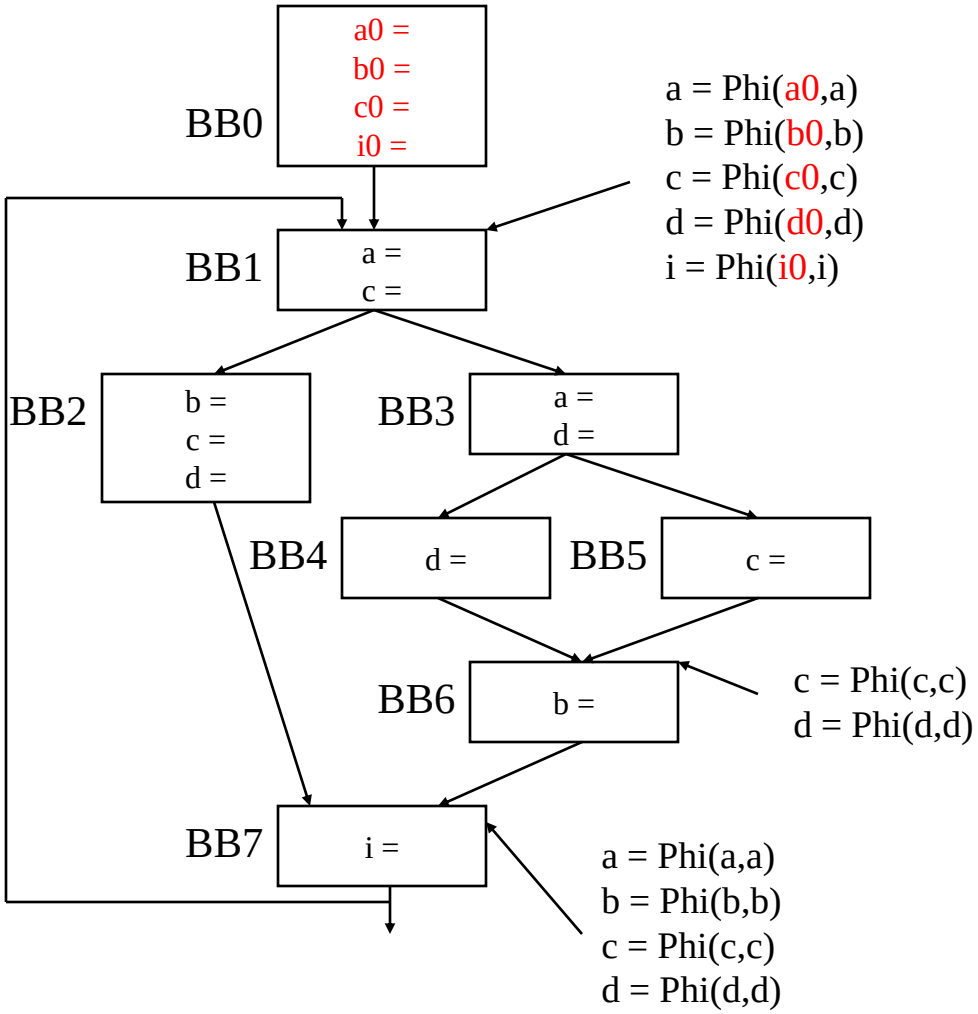
- Use an array of stacks, one stack per global variable (VR)
- Algorithm sketch
  - For each BB b in a preorder traversal of the dominator tree
    - Generate unique names for each Phi node
    - Rewrite each operation in the BB
      - Uses of global name: current name from stack
      - Defs of global name: create and push new name
    - Fill in Phi node parameters of successor blocks
    - Recurse on b's children in the dominator tree
    - <on exit from b> pop names generated in b from stacks

# Renaming – Example (Initial State)



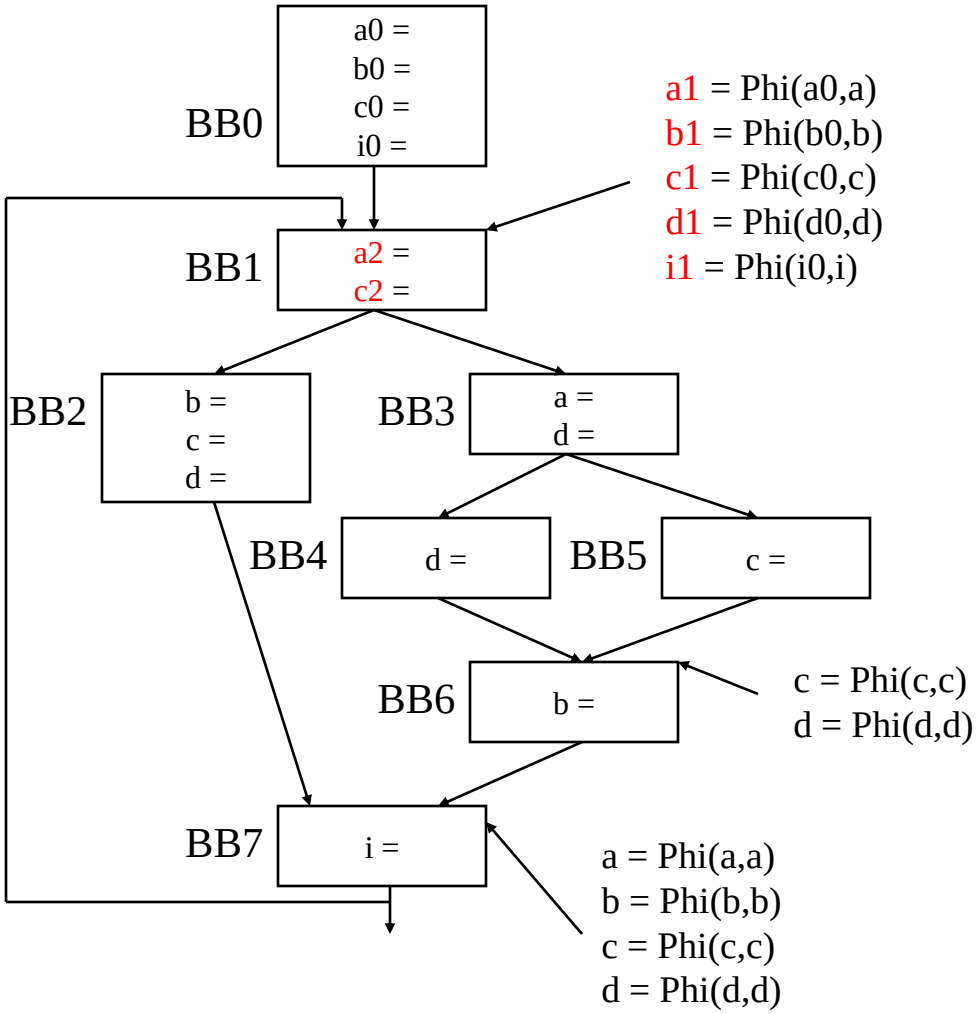

| var: | a | b | c | d | i |
|---|---|---|---|---|---|
| ctr: | 0 | 0 | 0 | 0 | 0 |
| stk: | a0 | b0 | c0 | d0 | i0 |

# Renaming – Example (After BB0)



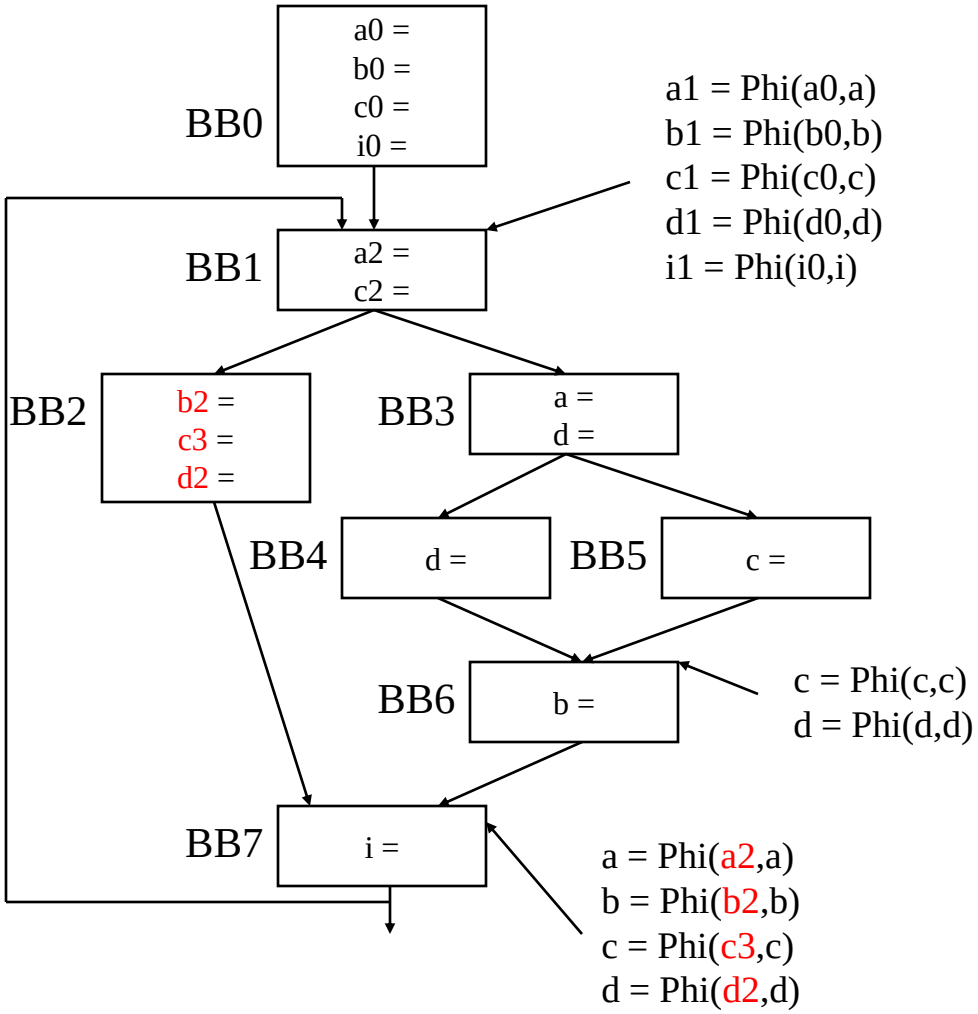

| var: | a | b | c | d | i |
|---|---|---|---|---|---|
| ctr: | 1 | 1 | 1 | 1 | 1 |
| stk: | a0 | b0 | c0 | d0 | i0 |

# Renaming – Example (After BB1)



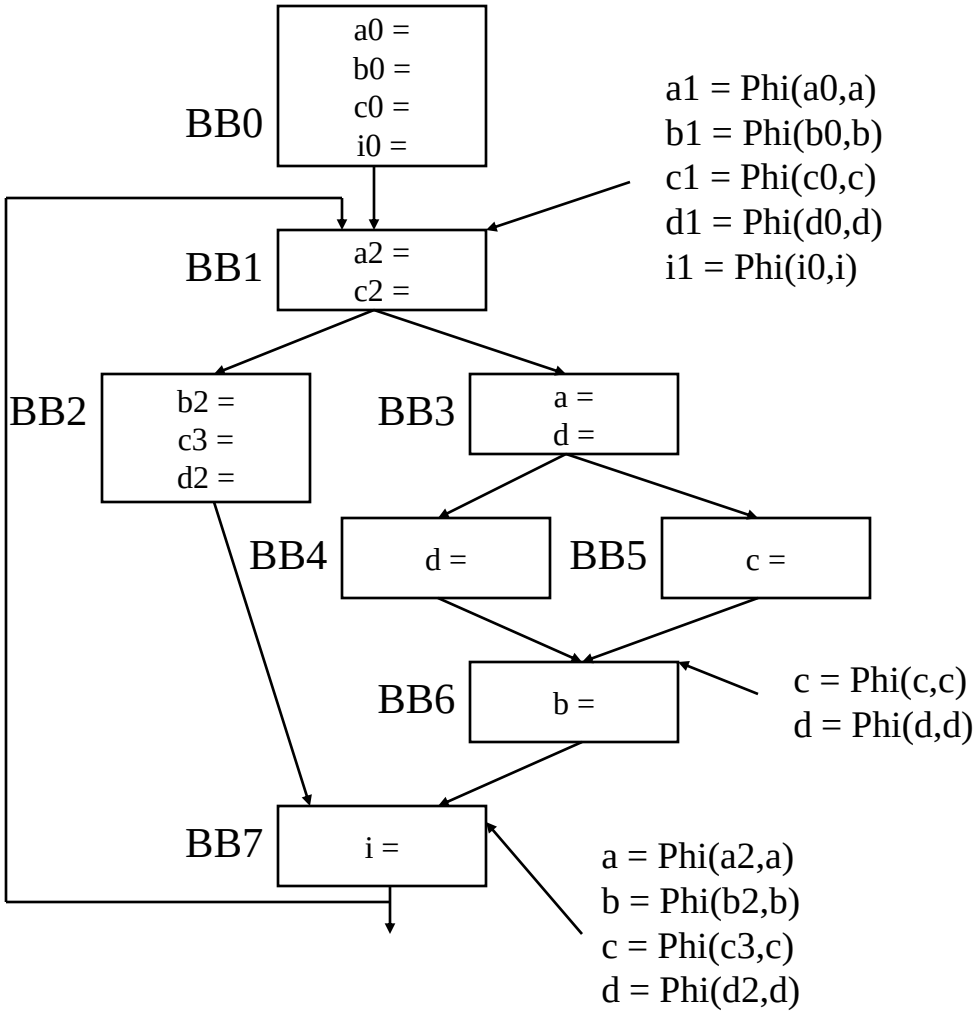

| var: | a | b | c | d | i |
|---|---|---|---|---|---|
| ctr: | 3 | 2 | 3 | 2 | 2 |
| stk: | a0 | b0 | c0 | d0 | i0 |
| | a1 | b1 | c1 | d1 | i1 |
| | a2 | | c2 | | |

# Renaming – Example (After BB2)




| var: | a | b | c | d | i |
|---|---|---|---|---|---|
| ctr: | 3 | 3 | 4 | 3 | 2 |
| stk: | a0 | b0 | c0 | d0 | i0 |
| | a1 | b1 | c1 | d1 | i1 |
| | a2 | b2 | c2 | d2 | |
| | | | c3 | | |

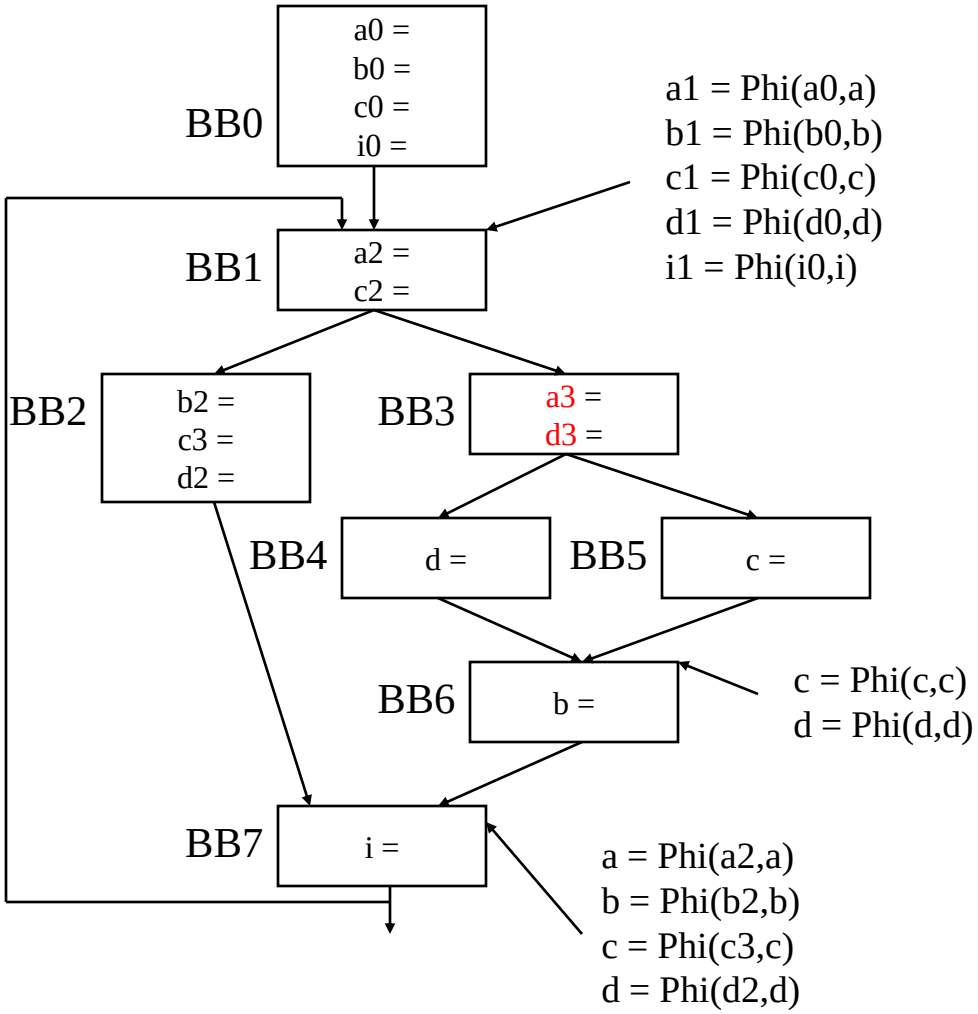
# Renaming – Example (Before BB3)




This just updates the stack to remove the stuff from the left path out of BB1

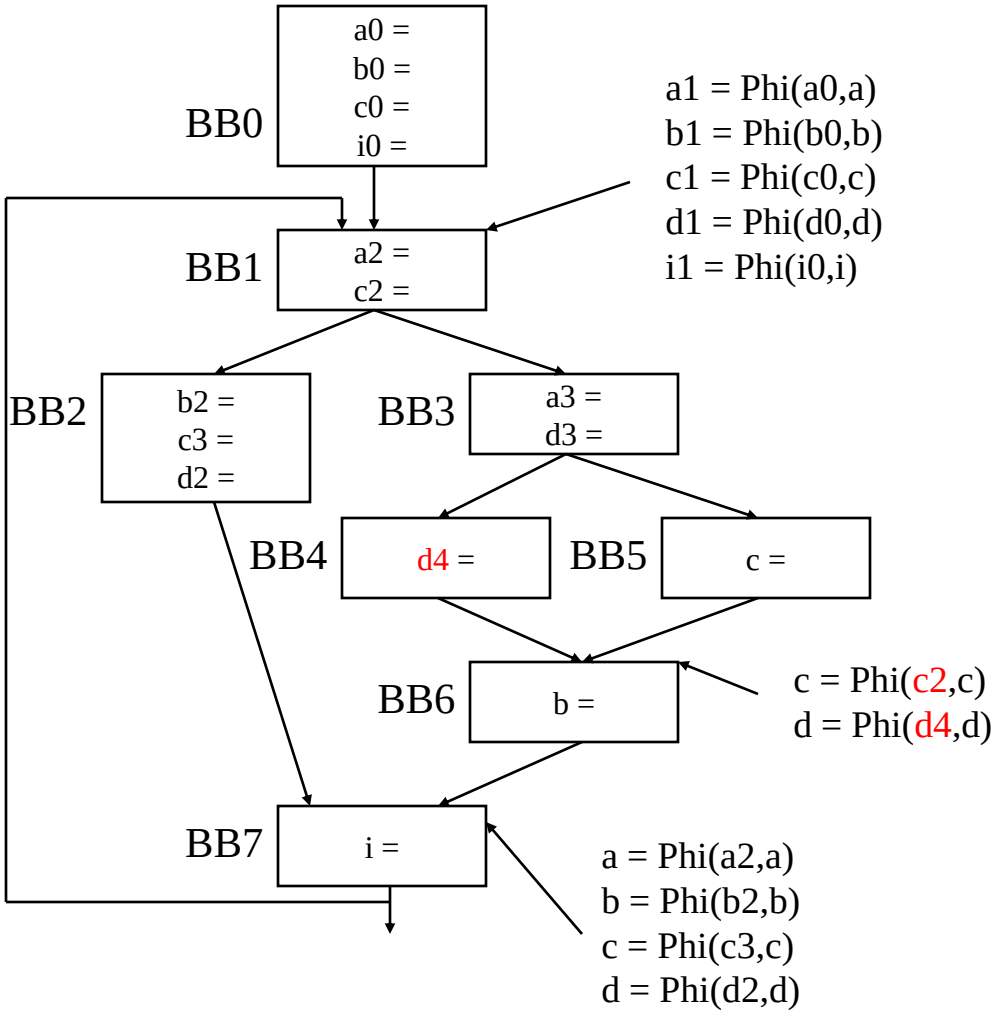
| var: | a | b | c | d | i |
|---|---|---|---|---|---|
| ctr: | 3 | 3 | 4 | 3 | 2 |
| stk: | a0 | b0 | c0 | d0 | i0 |
| | a1 | b1 | c1 | d1 | i1 |
| | a2 | | c2 | | |

# Renaming – Example (After BB3)



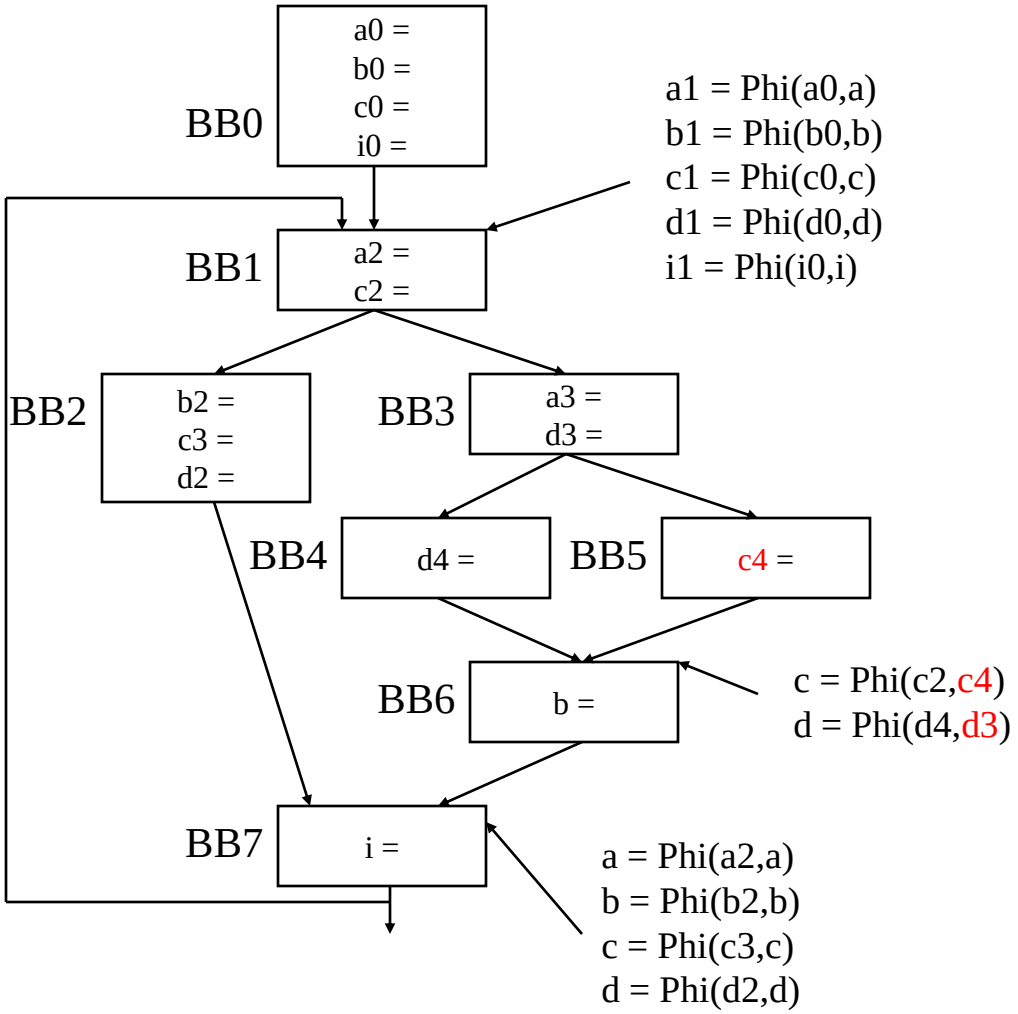

| var: | a | b | c | d | i |
|---|---|---|---|---|---|
| ctr: | 4 | 3 | 4 | 4 | 2 |
| stk: | a0 | b0 | c0 | d0 | i0 |
| | a1 | b1 | c1 | d1 | i1 |
| | a2 | | c2 | d3 | |
| | a3 | | | | |

# Renaming – Example (After BB4)



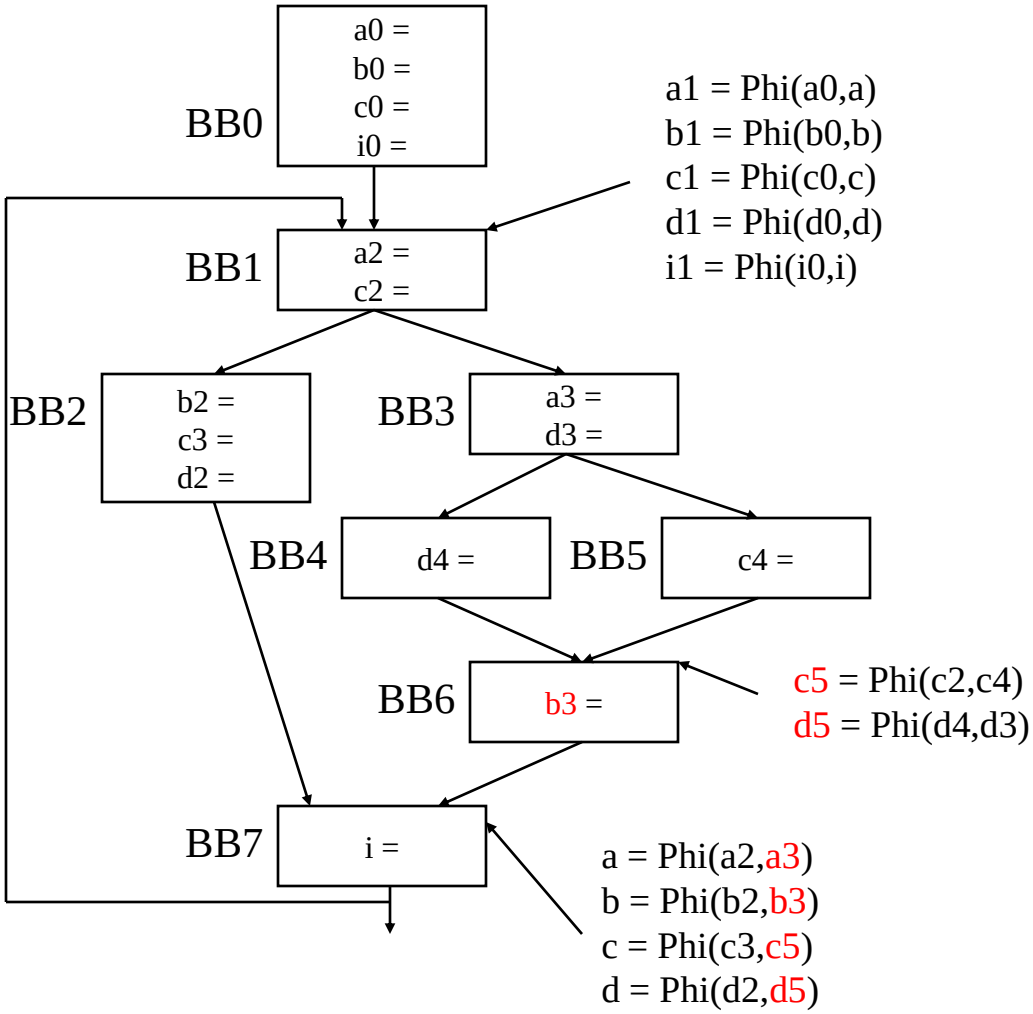

| var: | a | b | c | d | i |
|---|---|---|---|---|---|
| ctr: | 4 | 3 | 4 | 5 | 2 |
| stk: | a0 | b0 | c0 | d0 | i0 |
| | a1 | b1 | c1 | d1 | i1 |
| | a2 | | c2 | d3 | |
| | a3 | | | d4 | |

# Renaming – Example (After BB5)



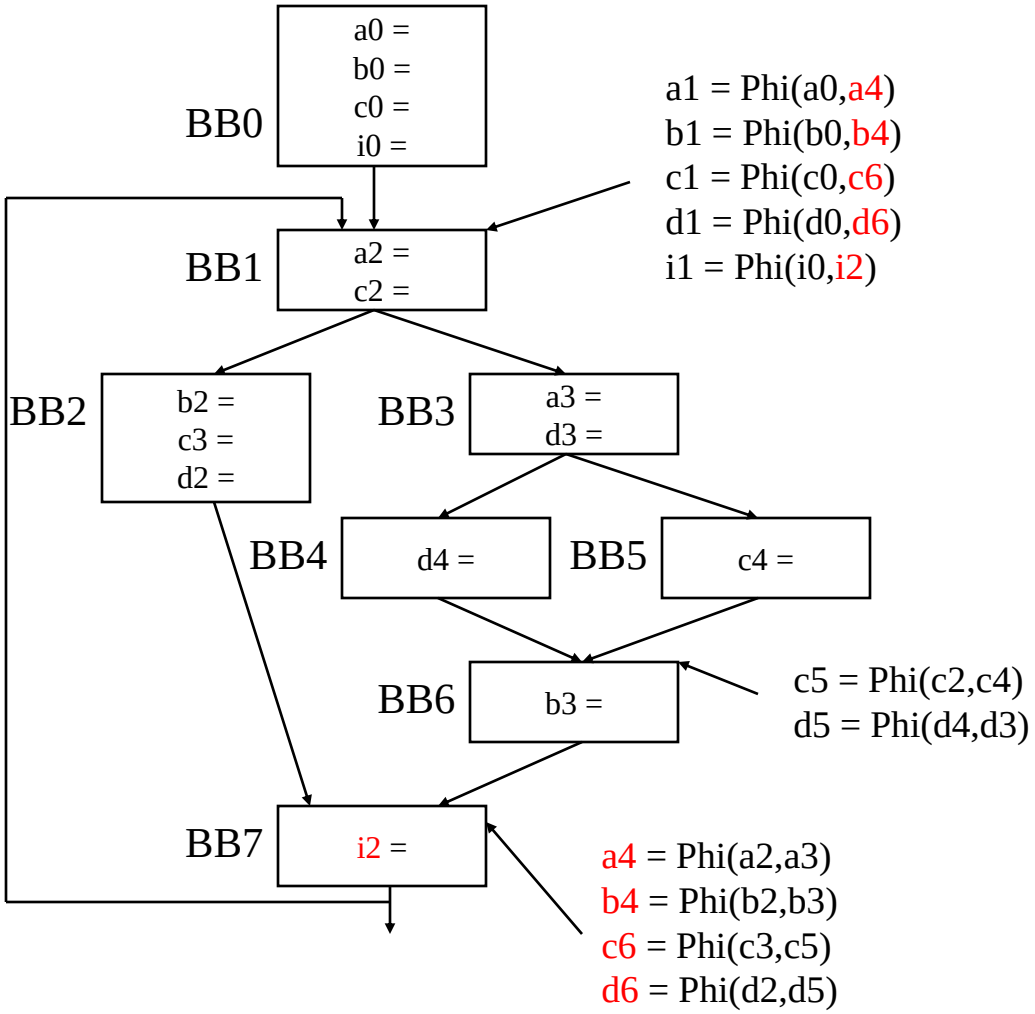

| var: | a | b | c | d | i |
|---|---|---|---|---|---|
| ctr: | 4 | 3 | 5 | 5 | 2 |
| stk: | a0 | b0 | c0 | d0 | i0 |
| | a1 | b1 | c1 | d1 | i1 |
| | a2 | | c2 | d3 | |
| | a3 | | c4 | | |

# Renaming – Example (After BB6)




| var: | a | b | c | d | i |
|---|---|---|---|---|---|
| ctr: | 4 | 4 | 6 | 6 | 2 |
| stk: | a0 | b0 | c0 | d0 | i0 |
| | a1 | b1 | c1 | d1 | i1 |
| | a2 | b3 | c2 | d3 | |
| | a3 | | c5 | d5 | |

# Renaming – Example (After BB7)




| var: | a | b | c | d | i |
|---|---|---|---|---|---|
| ctr: | 5 | 5 | 7 | 7 | 3 |
| stk: | a0 | b0 | c0 | d0 | i0 |
| | a1 | b1 | c1 | d1 | i1 |
| | a2 | b4 | c2 | d6 | i2 |
| | a4 | | c6 | | |

Fin!

# Class Problem

Rename the variables so this code is in SSA form