EECS 583 – Class 8

Static Single Assignment Form

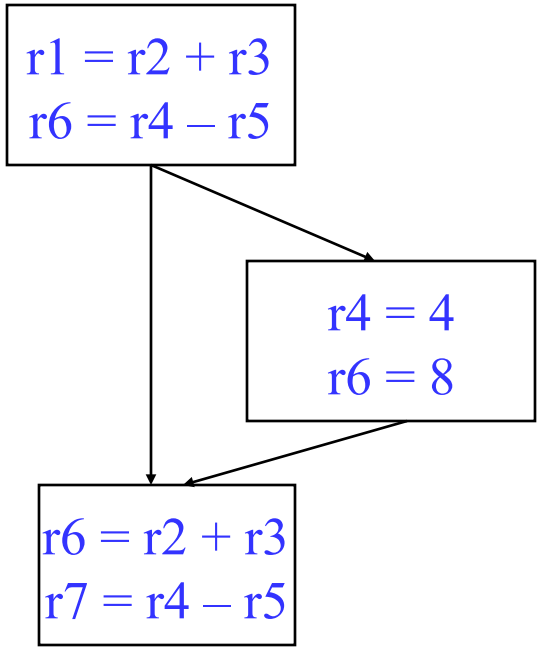
Classic Optimization

University of Michigan

September 29, 2014

Static Single Assignment (SSA) Form

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

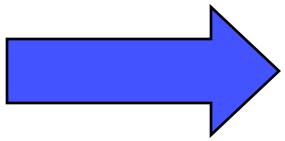


- Static single assignment
  - » Each assignment to a variable is given a unique name
  - » All of the uses reached by that assignment are renamed
  - » DU chains become obvious based on the register name!

Converting to SSA Form

- Trivial for straight line code

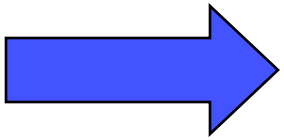
```
x = -1
y = x
x = 5
z = x
```



```
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
```

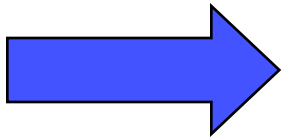


```
if ( ... )
   x0 = -1
else
   x1 = 5
x2 = Phi(x0,x1)
y = x2
```

Converting to SSA Form (2)

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

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



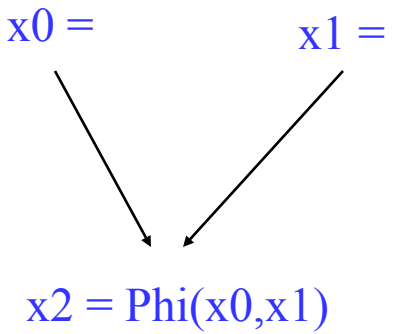
```
i0 = 0
do {
   i1 = Phi(i0, i2)
   i2 = i1 + 1
}
while (i2 < 50)
```

SSA Plusses and Minuses

- Advantages of SSA
  - » Explicit DU chains – Trivial to figure out what defs reach a use
    - Ÿ 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



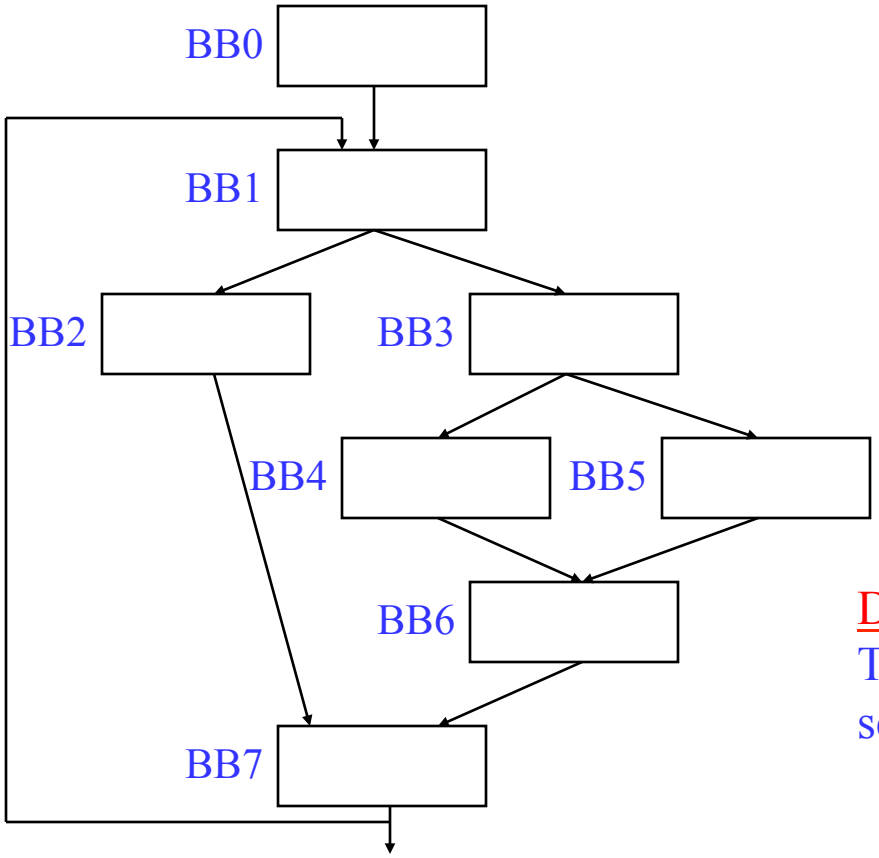
- Phi nodes are required when 2 non-null paths X→Z and Y→Z converge at node Z, and nodes X and Y contain assignments to V

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

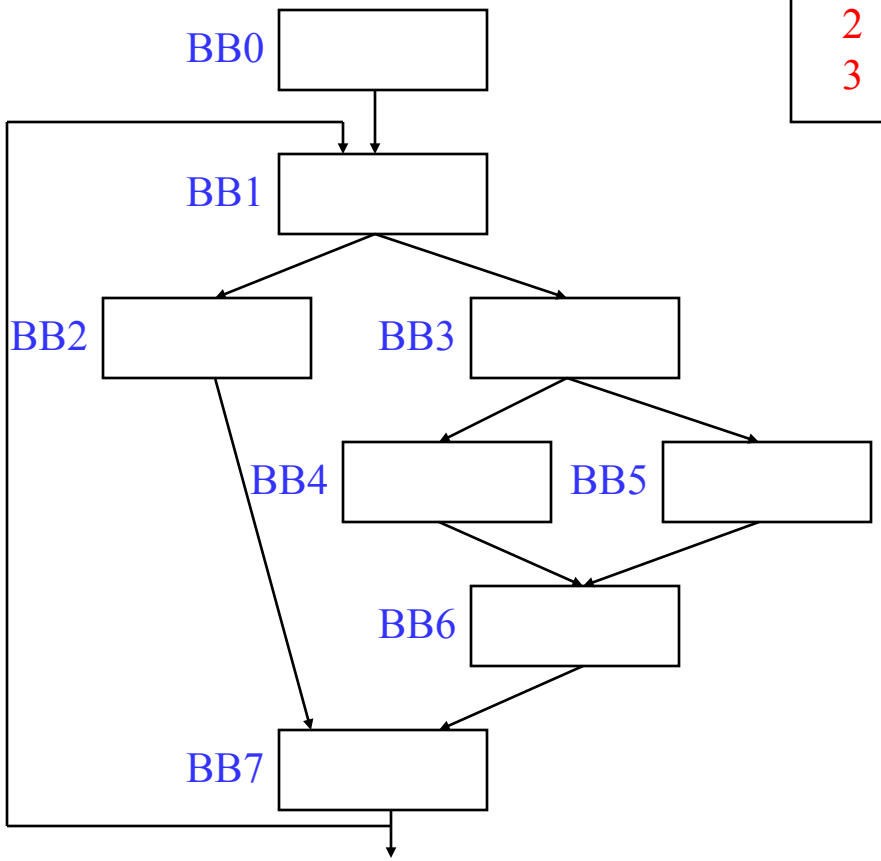
The dominance frontier of node X is the set of nodes Y such that

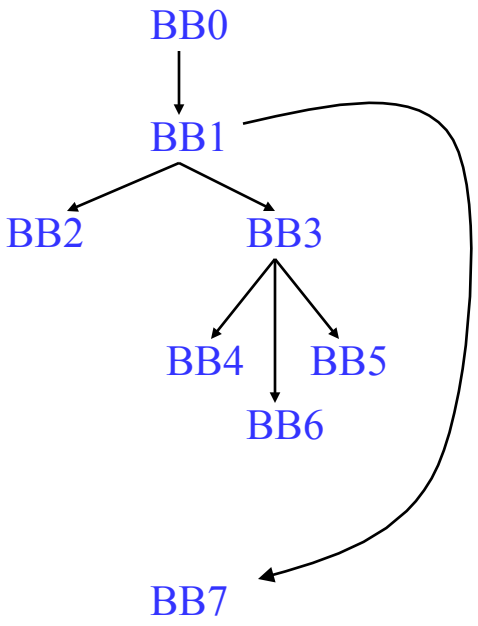
* X dominates a predecessor of Y, but
* X does not strictly dominate Y

Recall: Dominator Tree

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

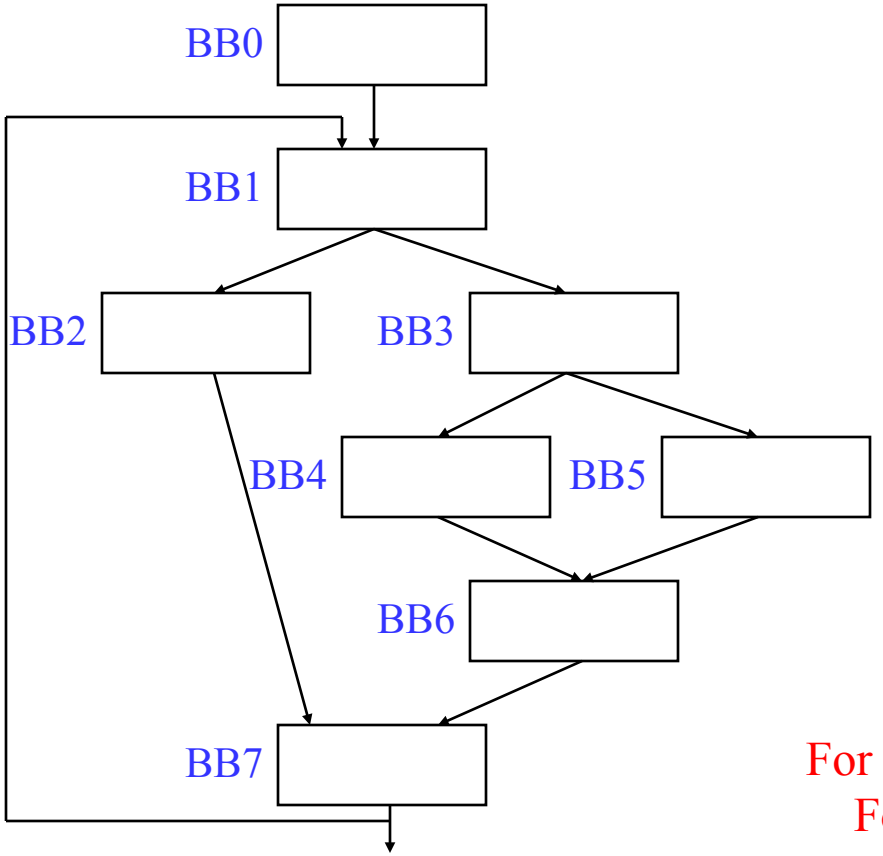
BB	DOM	BB	DOM
0	0	4	0,1,3,4
1	0,1	5	0,1,3,5
2	0,1,2	6	0,1,3,6
3	0,1,3	7	0,1,7

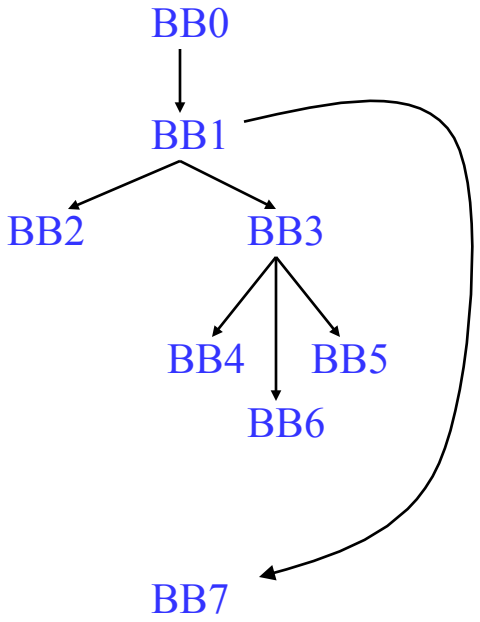




Dom tree

Computing Dominance Frontiers



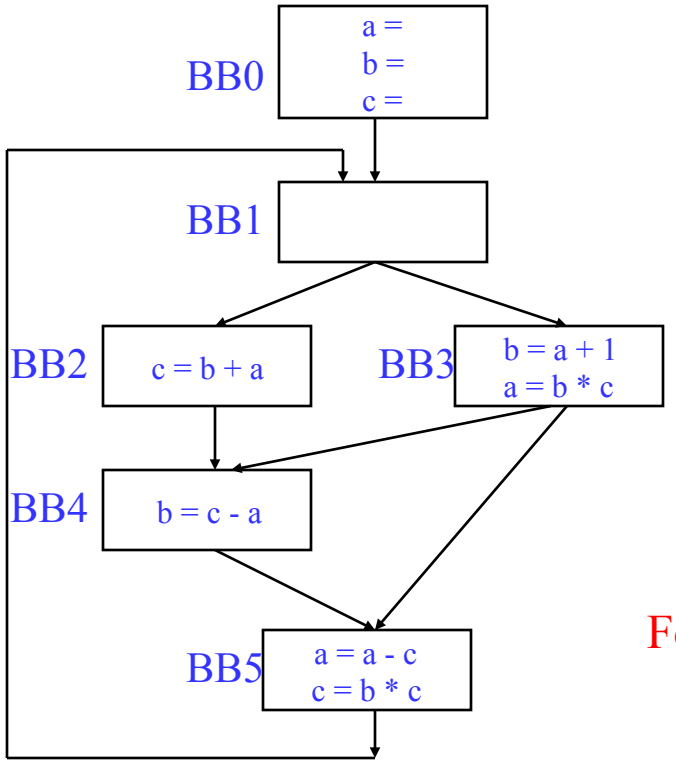


BB	DF
0	
1	
2	
3	
4	
5	
6	
7	

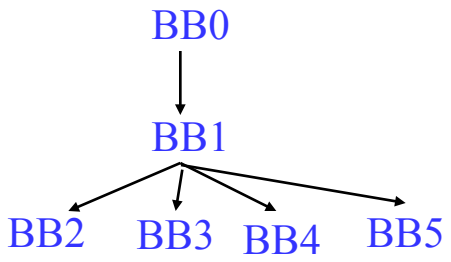
For each join point X in the CFG
  For each predecessor, Y, of X in the CFG
    Run up to the IDOM(X) in the dominator tree, adding X to DF(N) for each N between Y and IDOM(X) (or X, whichever is encountered first)

Class Problem

Compute dominance frontiers for each BB



Dominator Tree



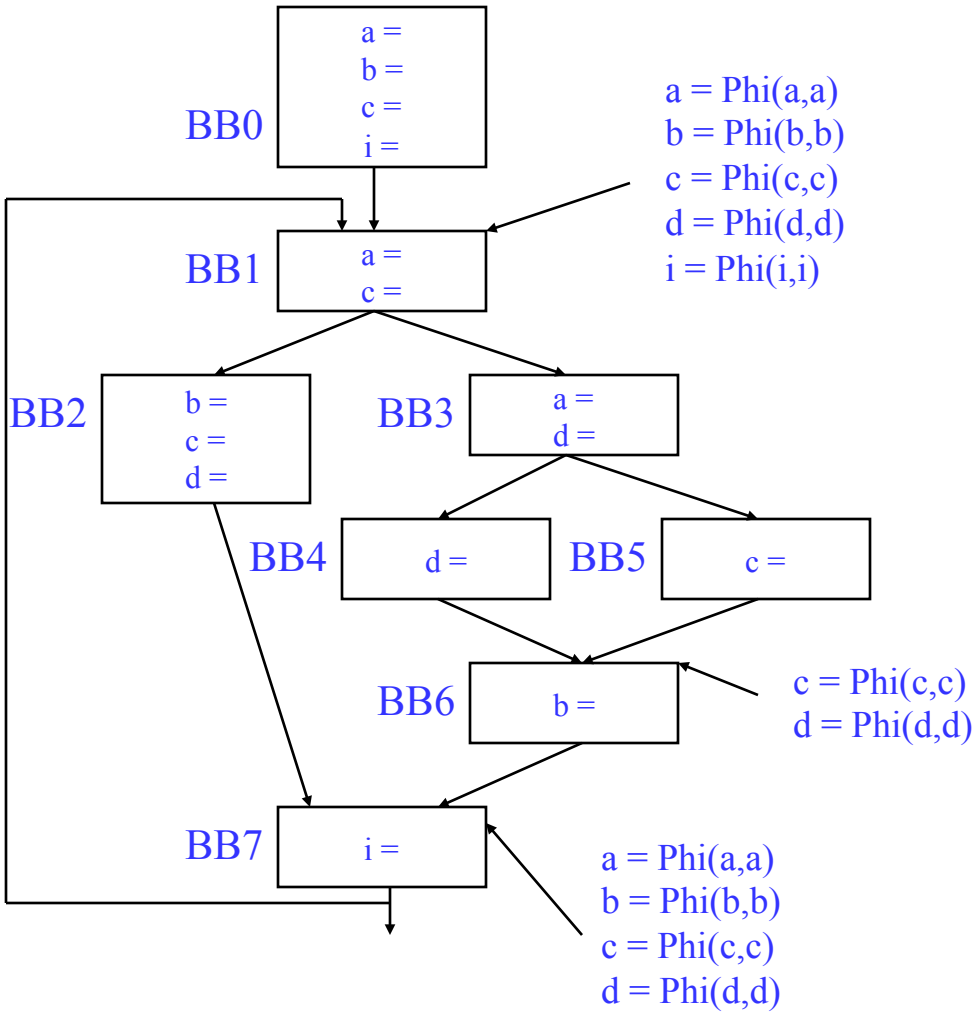
For each join point X in the CFG
  For each predecessor, Y, of X in the CFG
    Run up to the IDOM(X) in the dominator tree, adding X to DF(N) for each N between Y and IDOM(X) (or X, whichever is encountered first)

SSA Step 1 - Phi Node Insertion

- 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

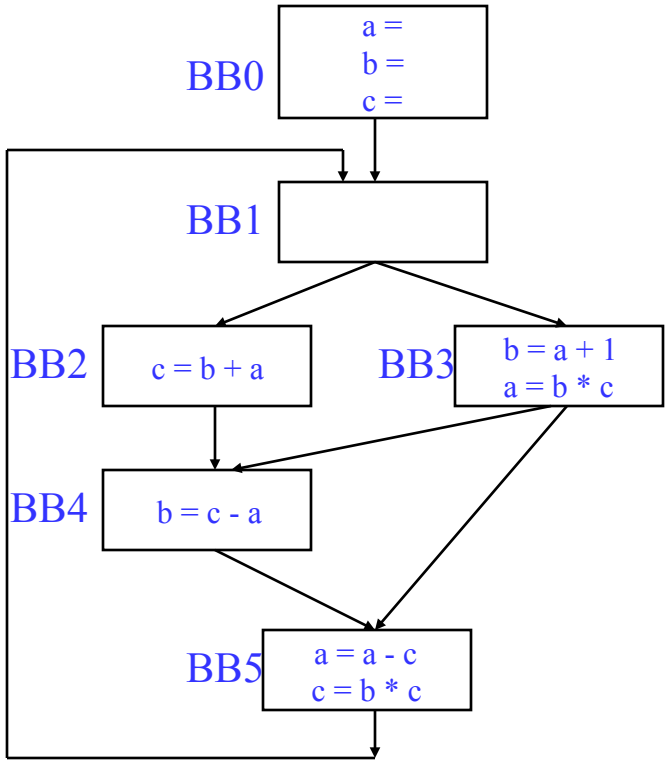
BB	DF
0	-
1	-
2	7
3	7
4	6
5	6
6	7
7	1



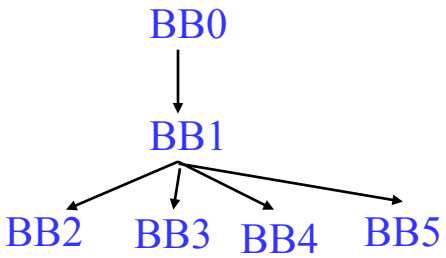
a is defined in 0,1,3
need Phi in 7
then a is defined in 7
need Phi in 1
b is defined in 0, 2, 6
need Phi in 7
then b is defined in 7
need Phi in 1
c is defined in 0,1,2,5
need Phi in 6,7
then c is defined in 7
need Phi in 1
d is defined in 2,3,4
need Phi in 6,7
then d is defined in 7
need Phi in 1
i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

Class Problem

Insert the Phi nodes



Dominator tree



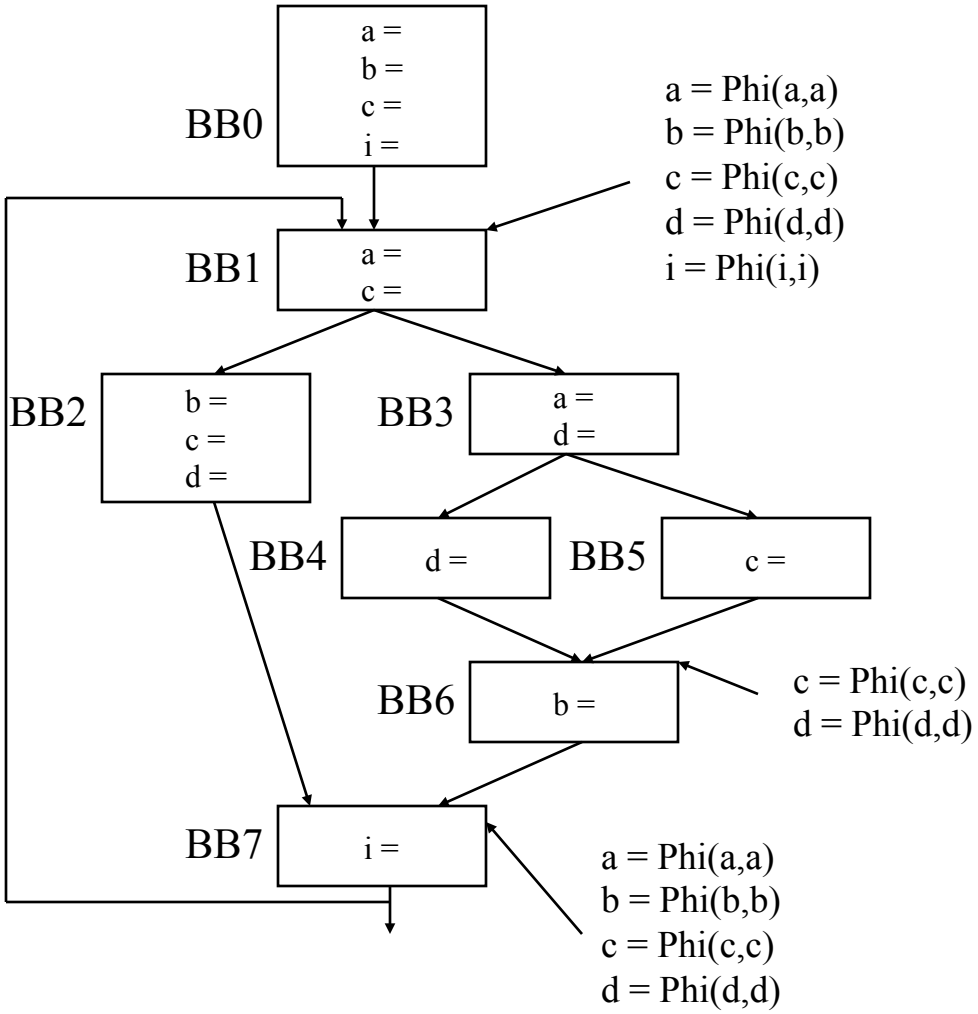
Dominance frontier

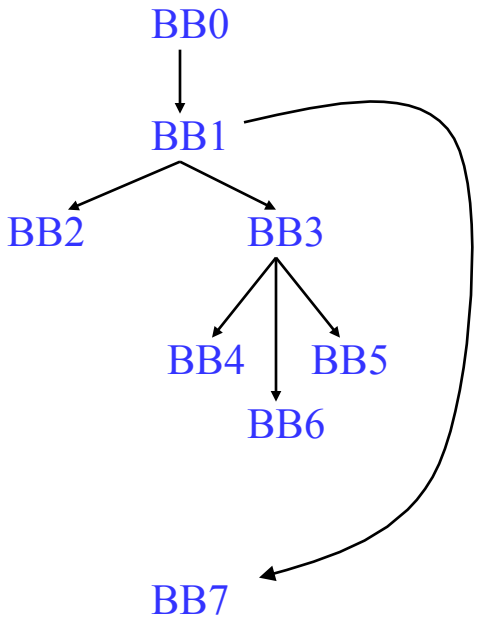
BB	DF
0	-
1	-
2	4
3	4, 5
4	5
5	1

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

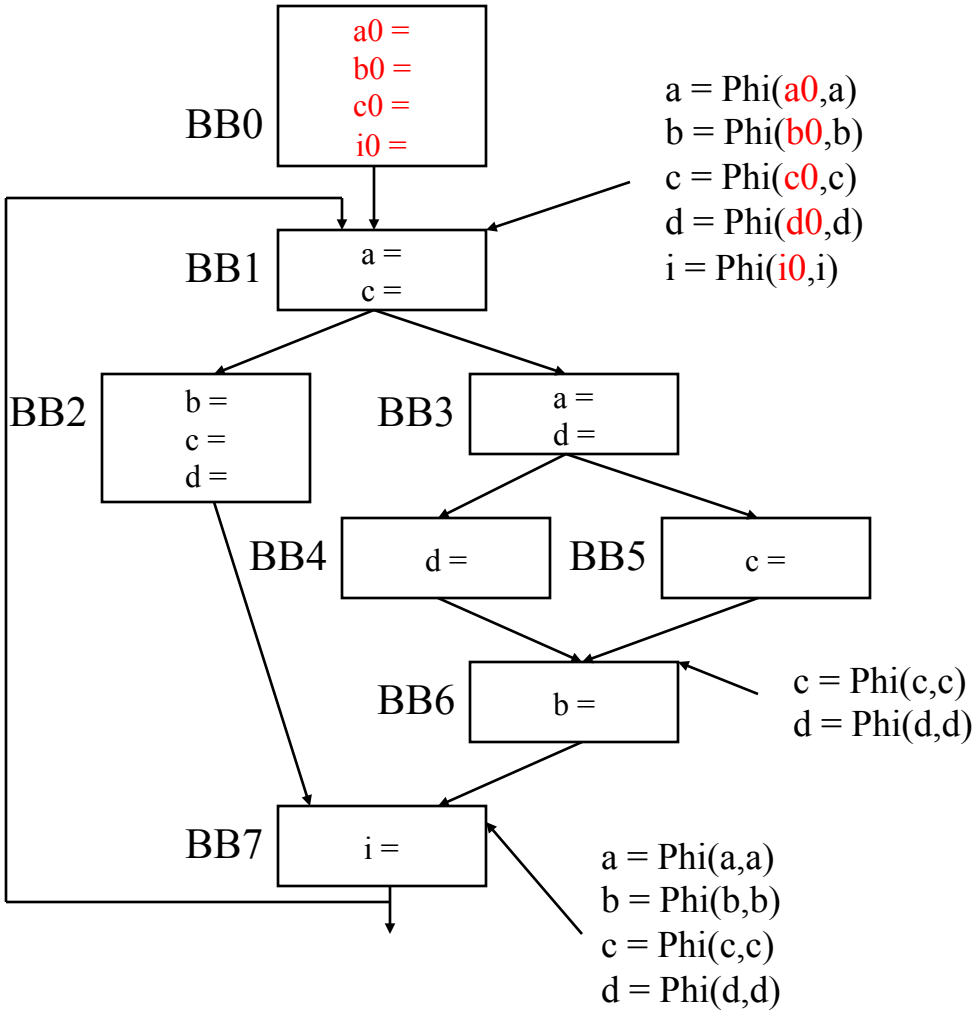
Renaming – Example (Initial State)

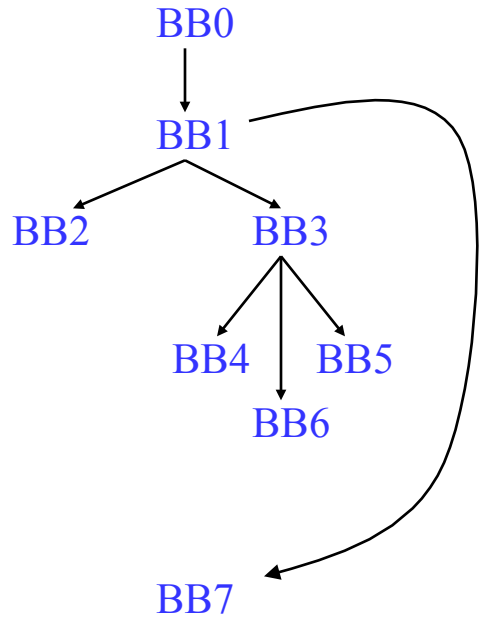




var:	a	b	c	d	i
ctr:	0	0	0	0	0
stk:	a0	b0	c0	d0	i0

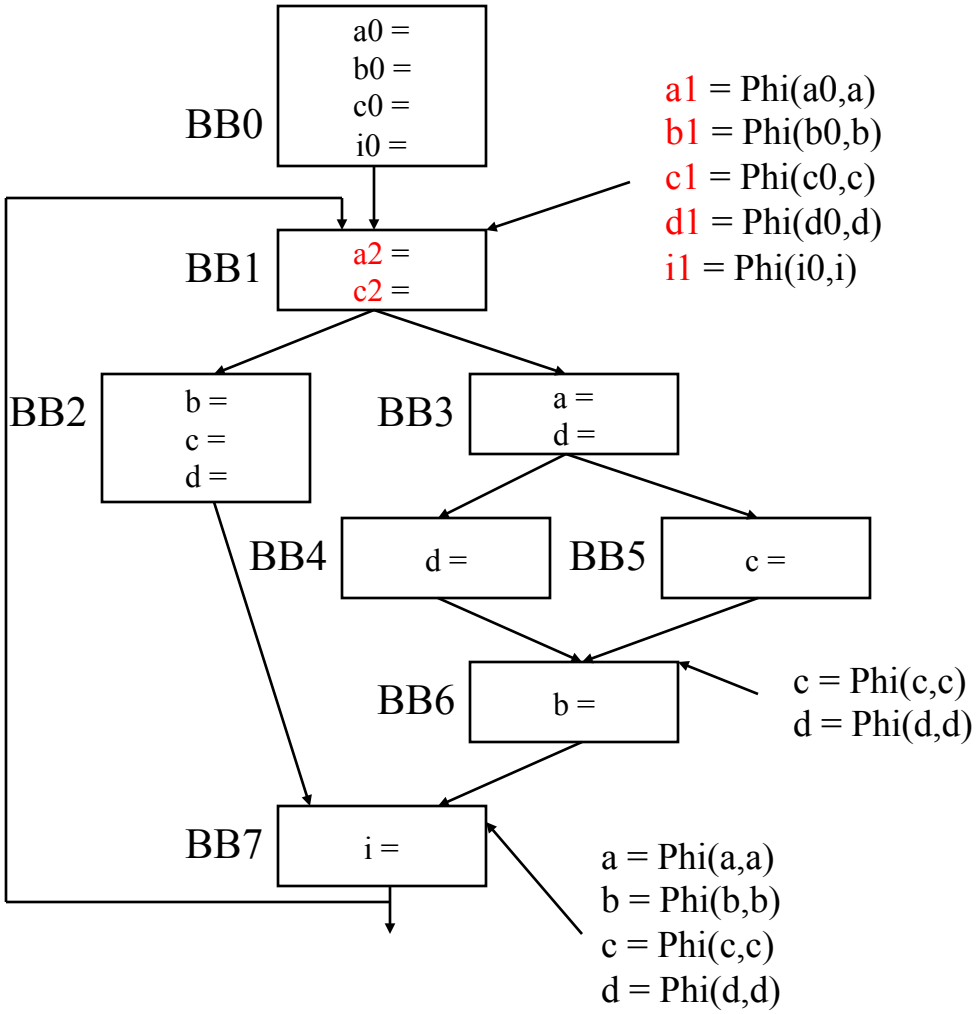
Renaming – Example (After BB0)

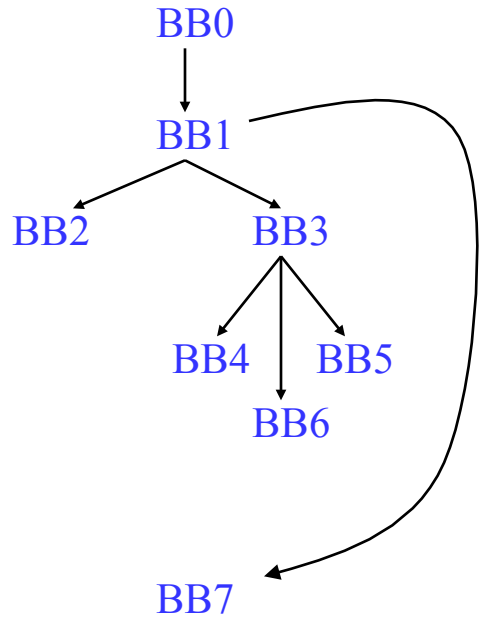




var:	a	b	c	d	i
ctr:	1	1	1	1	1
stk:	a0	b0	c0	d0	i0

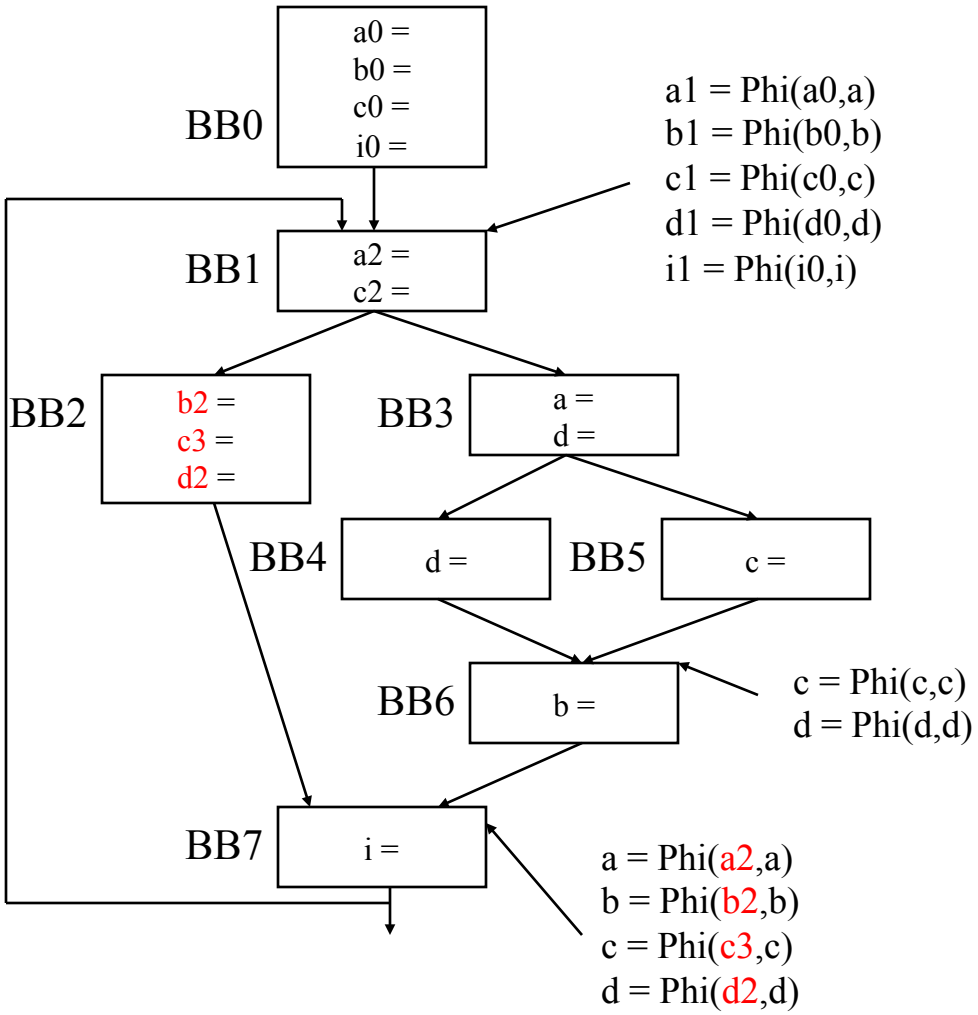
Renaming – Example (After BB1)

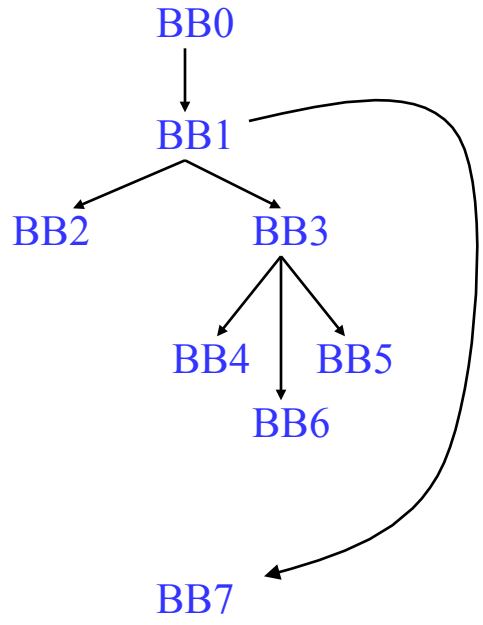




var:	a	b	c	d	i
ctr:	3	2	3	2	2
stk:	a0	b0	c0	d0	i0
	a1	b1	c1	d1	i1
	a2		c2		

Renaming – Example (After BB2)

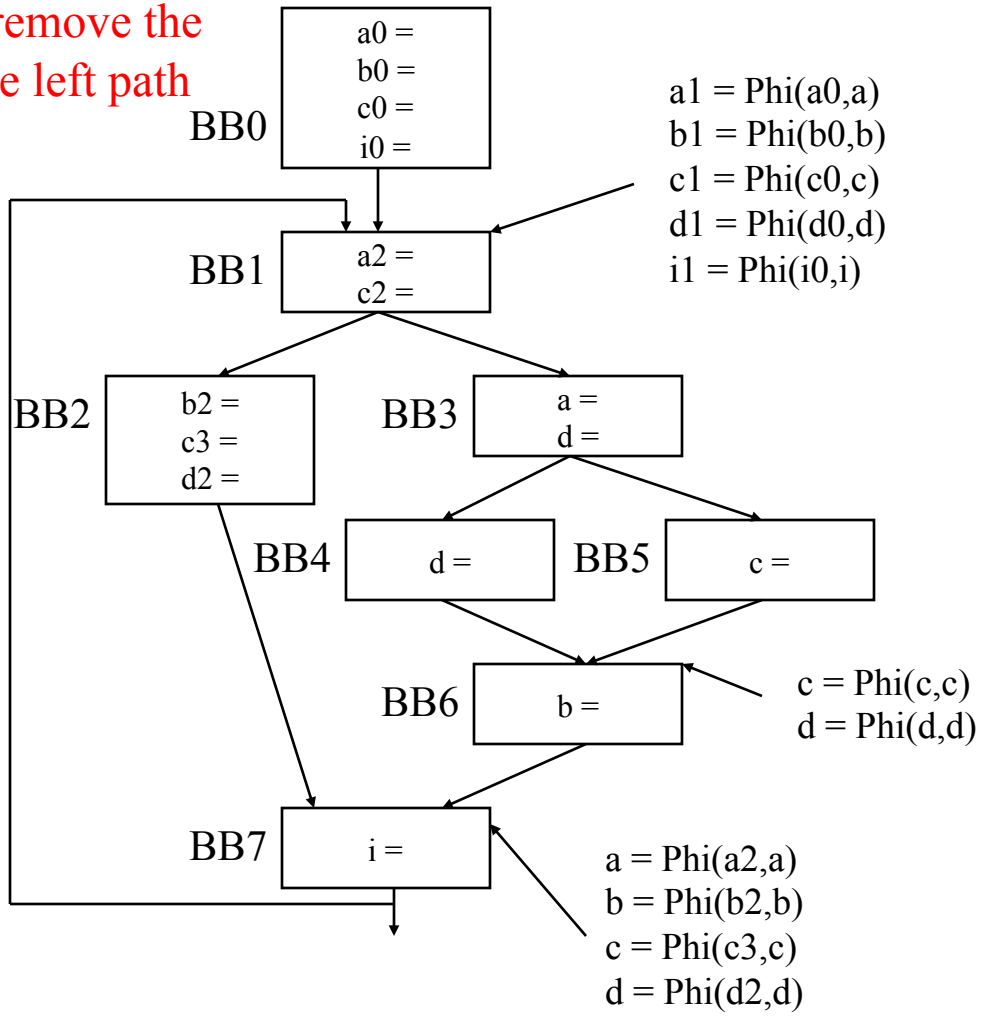


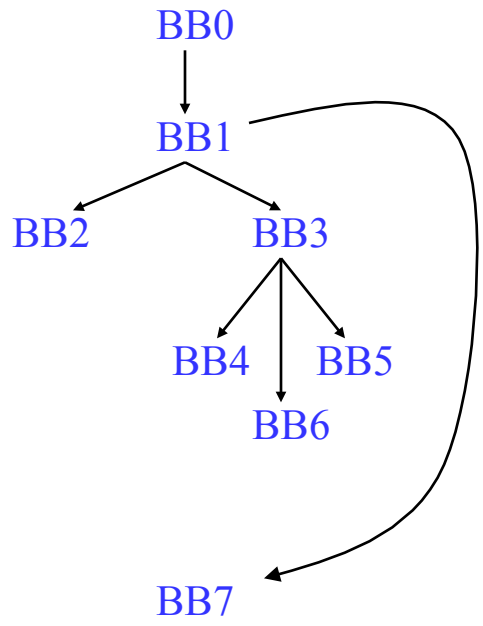


var:	a	b	c	d	i
ctr:	3	3	4	3	2
stk:	a0	b0	c0	d0	i0
	a1	b1	c1	d1	i1
	a2	b2	c2	d2	
			c3		

Renaming – Example (Before BB3)

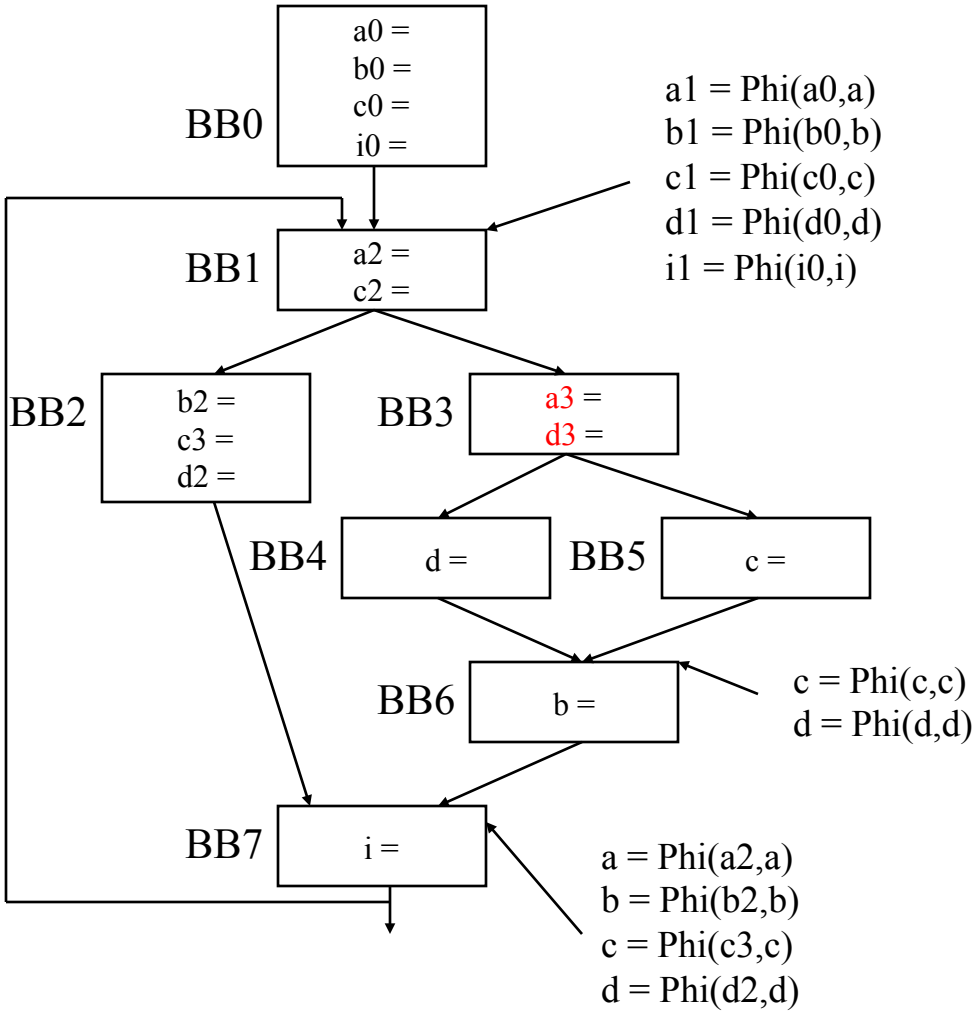
This just updates the stack to remove the stuff from the left path out of BB1

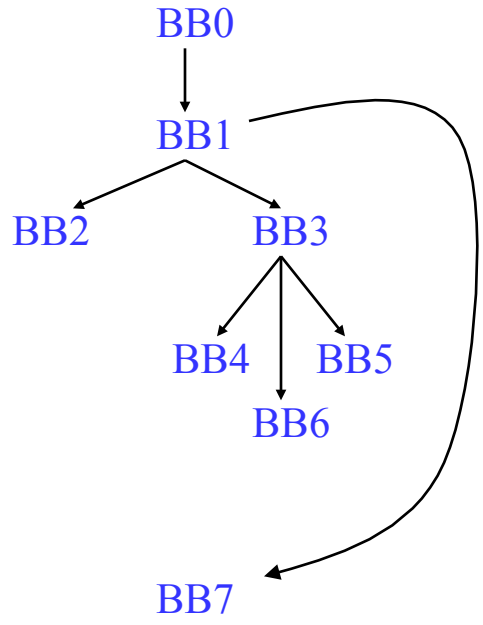




var:	a	b	c	d	i
ctr:	3	3	4	3	2
stk:	a0	b0	c0	d0	i0
	a1	b1	c1	d1	i1
	a2		c2		

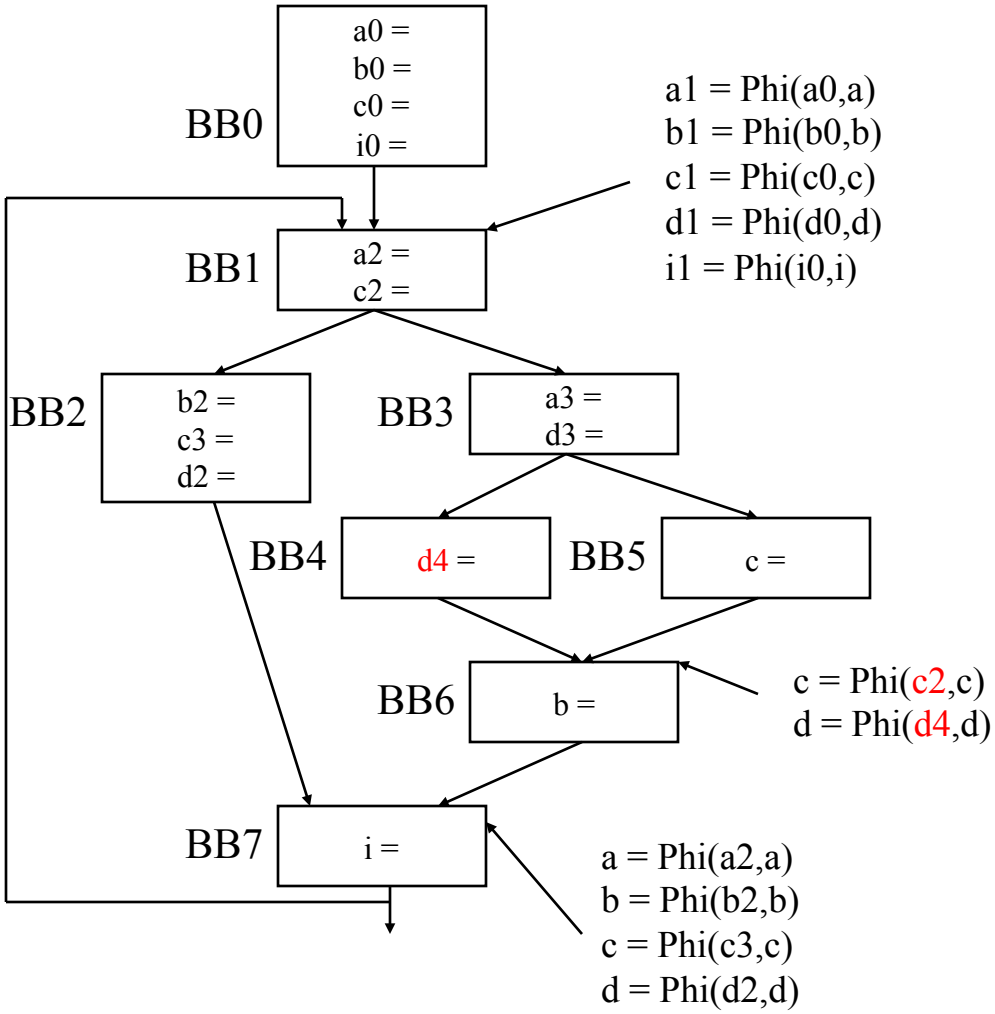
Renaming – Example (After BB3)

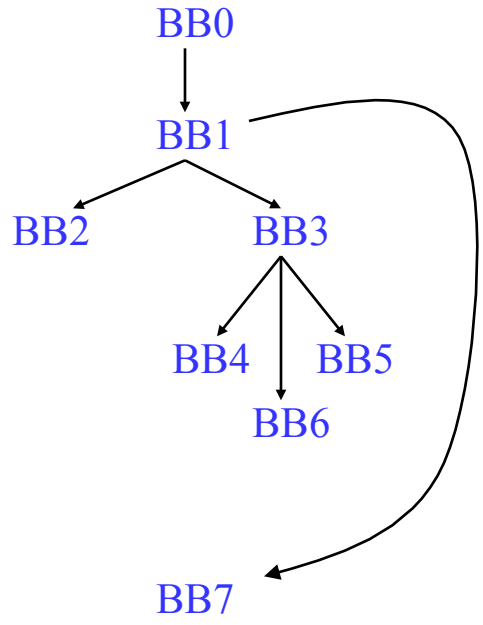




var:	a	b	c	d	i
ctr:	4	3	4	4	2
stk:	a0	b0	c0	d0	i0
	a1	b1	c1	d1	i1
	a2		c2	d3	
	a3				

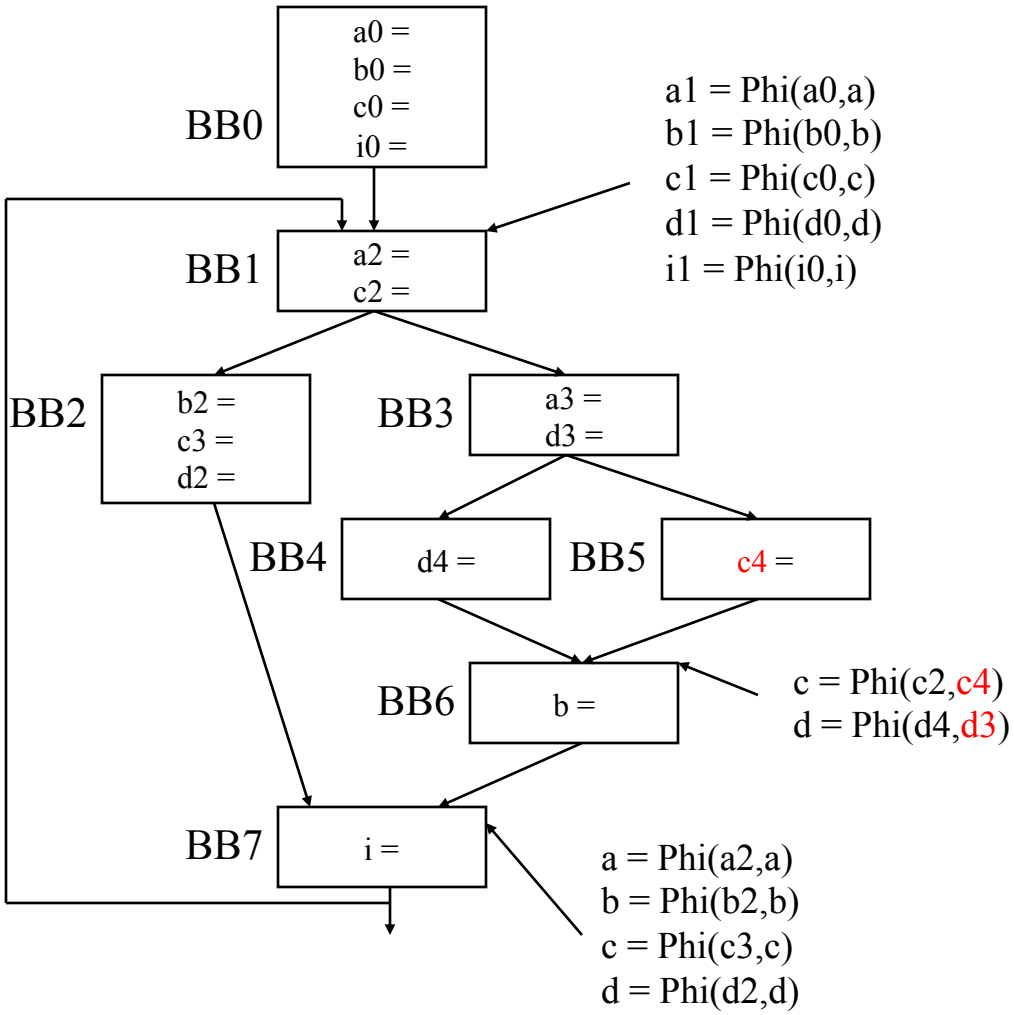
Renaming – Example (After BB4)

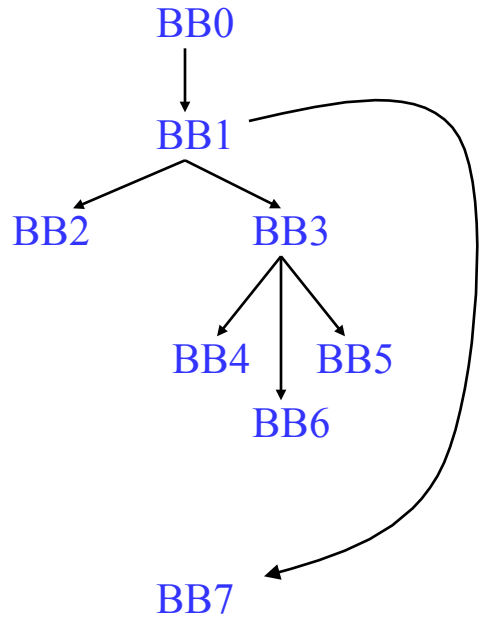




var:	a	b	c	d	i
ctr:	4	3	4	5	2
stk:	a0	b0	c0	d0	i0
	a1	b1	c1	d1	i1
	a2		c2	d3	
	a3			d4	

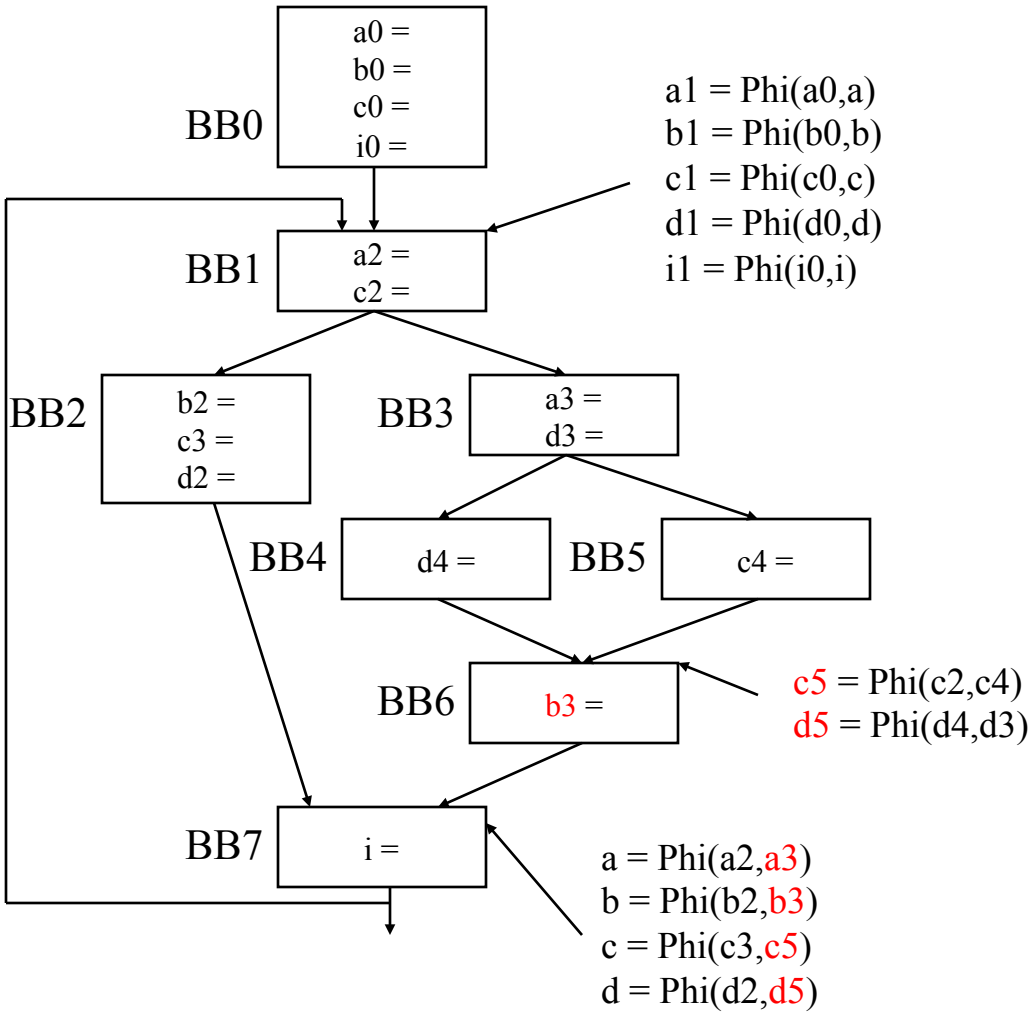
Renaming – Example (After BB5)

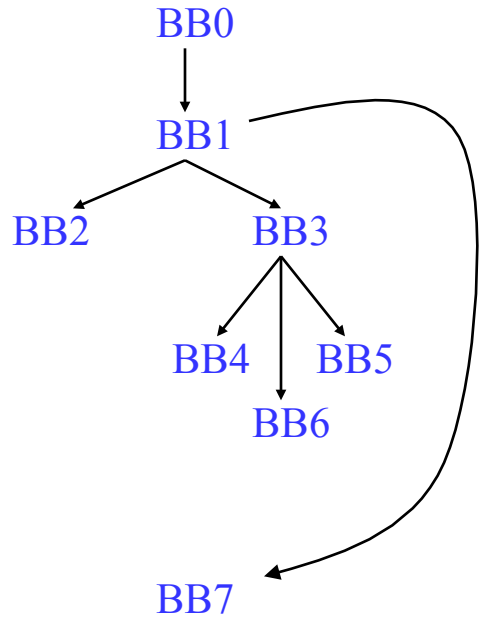




var:	a	b	c	d	i
ctr:	4	3	5	5	2
stk:	a0	b0	c0	d0	i0
	a1	b1	c1	d1	i1
	a2		c2	d3	
	a3		c4		

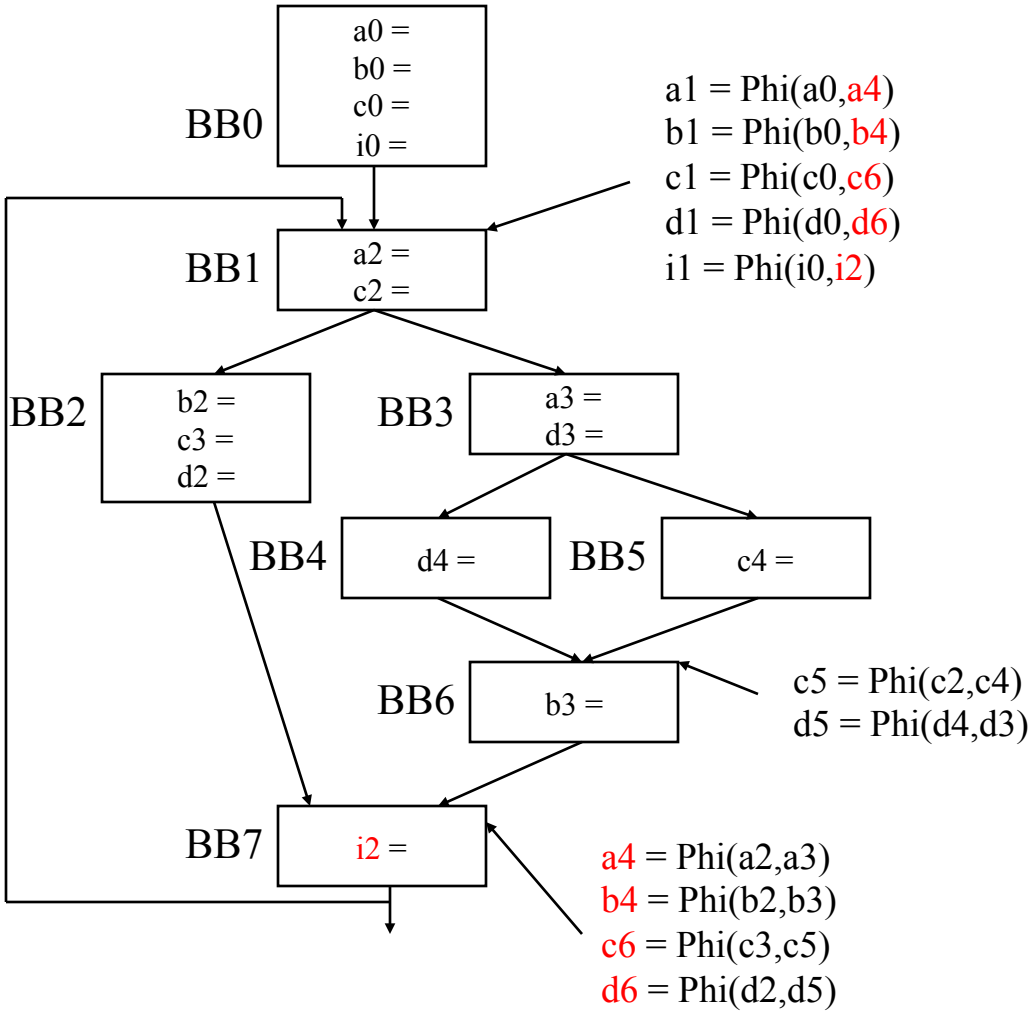
Renaming – Example (After BB6)

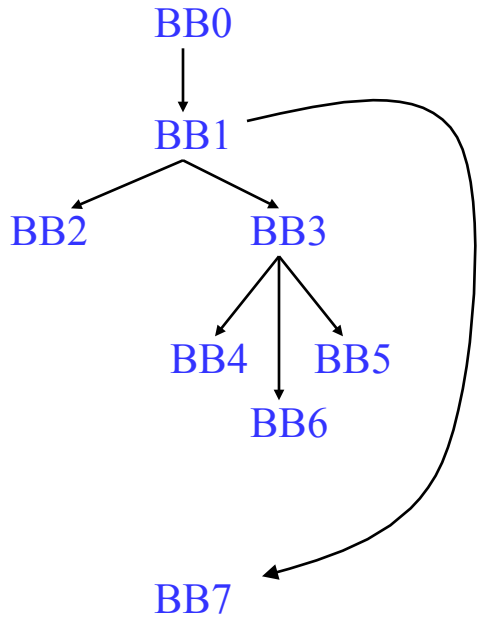




var:	a	b	c	d	i
ctr:	4	4	6	6	2
stk:	a0	b0	c0	d0	i0
	a1	b1	c1	d1	i1
	a2	b3	c2	d3	
	a3		c5	d5	

Renaming – Example (After BB7)



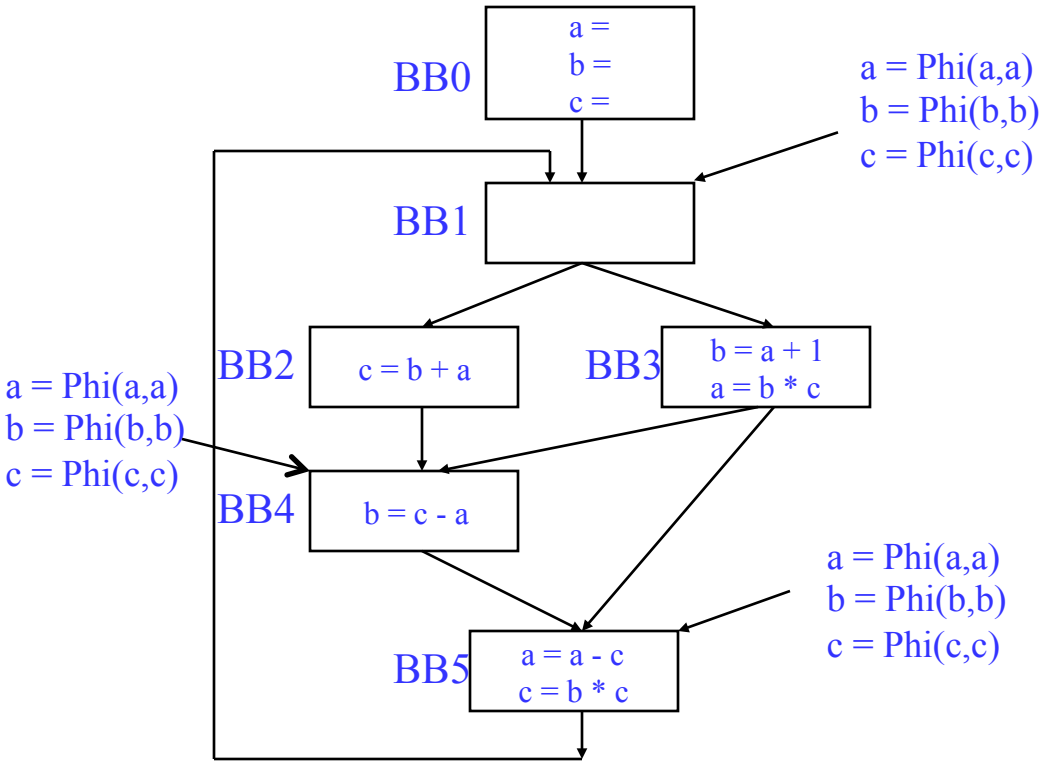


var:	a	b	c	d	i
ctr:	5	5	7	7	3
stk:	a0	b0	c0	d0	i0
	a1	b1	c1	d1	i1
	a2	b4	c2	d6	i2
	a4		c6		

Fin!

Class Problem

Rename the variables



Dominance frontier

BB	DF
0	-
1	-
2	4
3	4, 5
4	5
5	1

Code Optimization

Code Optimization

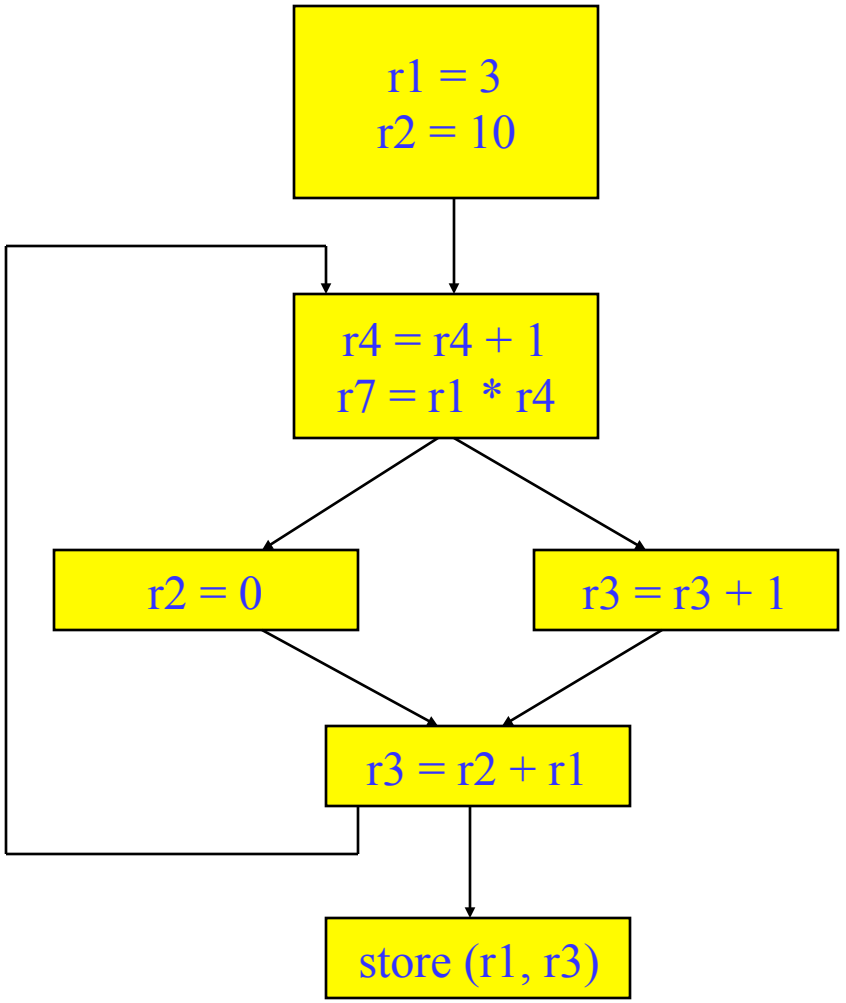
- Make the code run faster on the target processor
  - Other objectives: Power, code size
- Classes of optimization
  - 1. Classical (machine independent)
    - Reducing operation count (redundancy elimination)
    - Simplifying operations
    - Generally good for any kind of machine
  - 2. Machine specific
    - Peephole optimizations
    - Take advantage of specialized hardware features
  - 3. Parallelism enhancing
    - Increasing parallelism (ILP or TLP)
    - Possibly increase instructions

A Tour Through the Classical Optimizations

- For this class – Go over concepts of a small subset of the optimizations
  - What it is, why its useful
  - When can it be applied (set of conditions that must be satisfied)
  - How it works
  - Give you the flavor but don't want to beat you over the head
- Challenges
  - Register pressure?
  - Parallelism verses operation count

Dead Code Elimination

- Remove any operation who's result is never consumed
- Rules
  - X can be deleted
    - no stores or branches
  - DU chain empty or dest register not live
- This misses some dead code!!
  - Especially in loops
  - Critical operation
    - store or branch operation
  - Any operation that does not directly or indirectly feed a critical operation is dead
  - Trace UD chains backwards from critical operations
  - Any op not visited is dead



Local Constant Propagation

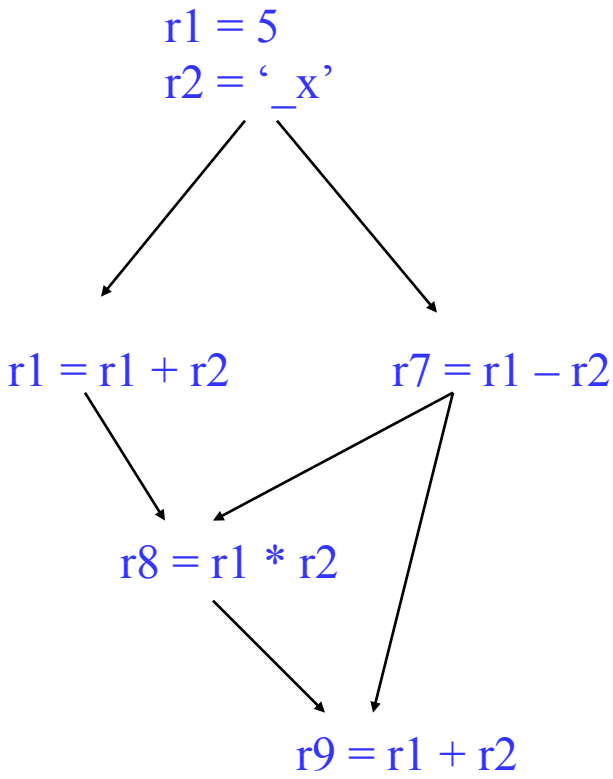
- Forward propagation of moves of the form
  - » rx = L (where L is a literal)
  - » Maximally propagate
- Consider 2 ops, X and Y in a BB, X is before Y
  - » 1. X is a move
  - » 2. src1(X) is a literal
  - » 3. Y consumes dest(X)
  - » 4. There is no definition of dest(X) between X and Y
  - » 5. No danger betw X and Y
    - Ÿ When dest(X) is a Macro reg, BRL destroys the value

```
r1 = 5
r2 = '_x'
r3 = 7
r4 = r4 + r1
r1 = r1 + r2
r1 = r1 + 1
r3 = 12
r8 = r1 - r2
r9 = r3 + r5
r3 = r2 + 1
r10 = r3 – r1
```

Note, ignore operation format issues, so all operations can have literals in either operand position

Global Constant Propagation

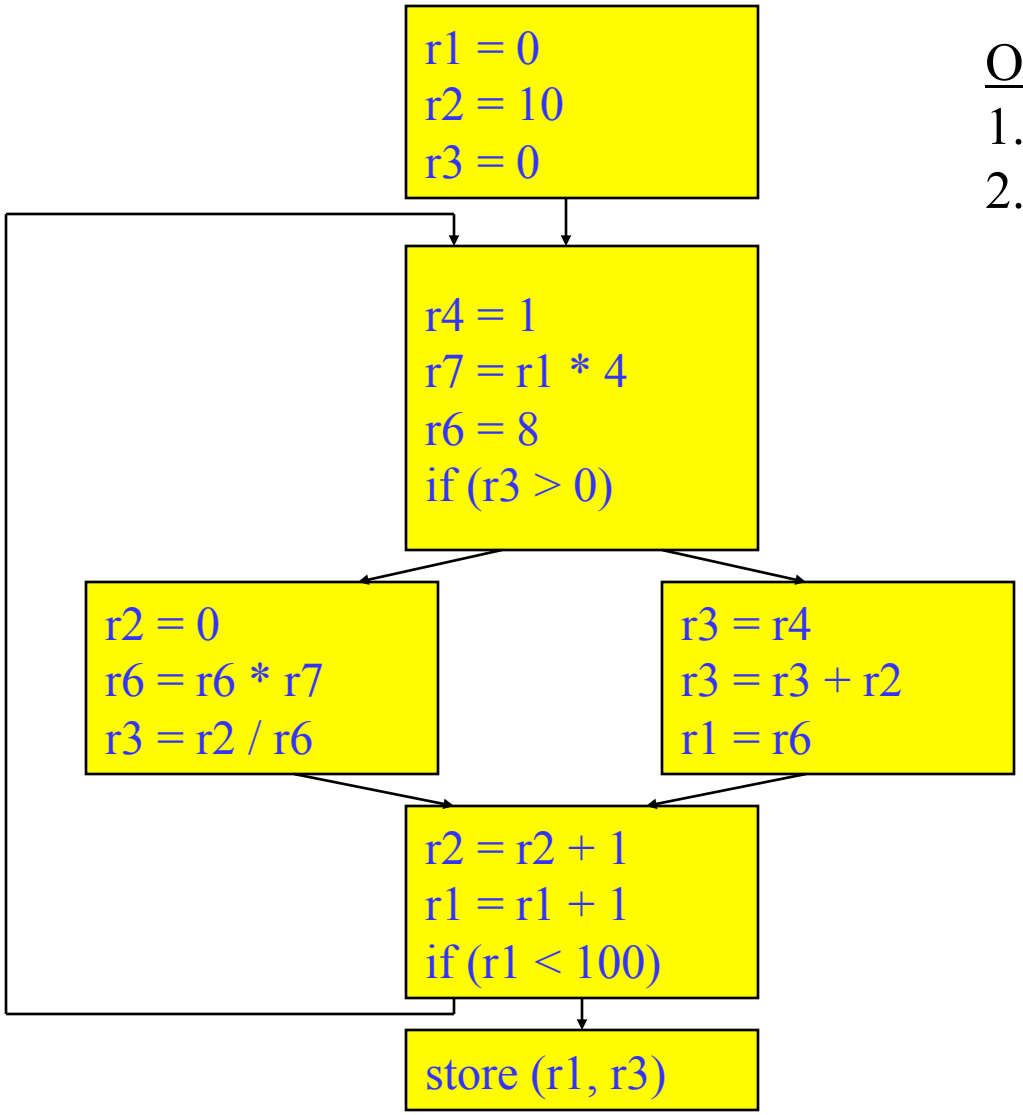
- Consider 2 ops, X and Y in different BBs
  - 1. X is a move
  - 2. src1(X) is a literal
  - 3. Y consumes dest(X)
  - 4. X is in a_in(BB(Y))
  - 5. Dest(x) is not modified between the top of BB(Y) and Y
  - 6. No danger betw X and Y
    - When dest(X) is a Macro reg, BRL destroys the value



Constant Folding

- Simplify 1 operation based on values of src operands
  - Constant propagation creates opportunities for this
- All constant operands
  - Evaluate the op, replace with a move
    - r1 = 3 * 4 → r1 = 12
    - r1 = 3 / 0 → ??? Don't evaluate excepting ops !, what about floating-point?
  - Evaluate conditional branch, replace with BRU or noop
    - if (1 < 2) goto BB2 → BRU BB2
    - if (1 > 2) goto BB2 → convert to a noop
- Algebraic identities
  - r1 = r2 + 0, r2 – 0, r2 | 0, r2 ^ 0, r2 << 0, r2 >> 0
    - r1 = r2
  - r1 = 0 * r2, 0 / r2, 0 & r2
    - r1 = 0
  - r1 = r2 * 1, r2 / 1
    - r1 = r2

Class Problem

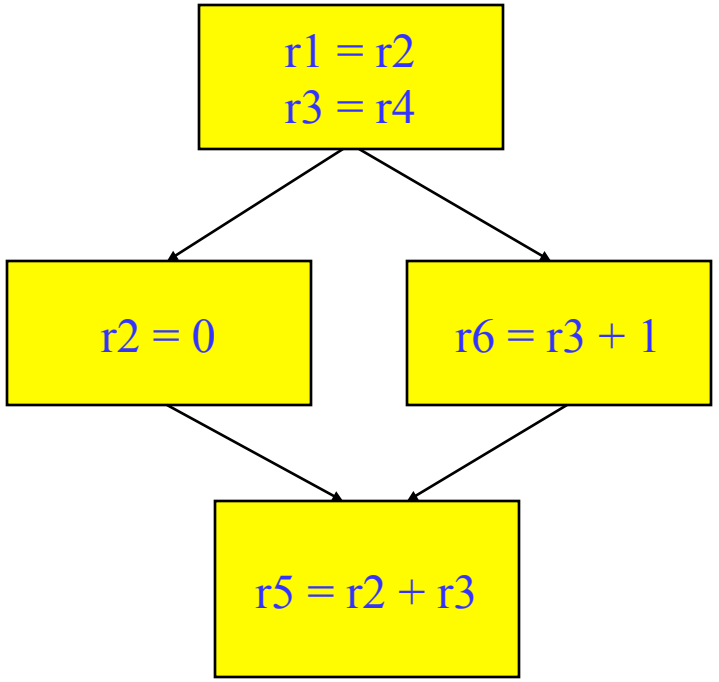


Optimize this applying

1. constant propagation
2. constant folding

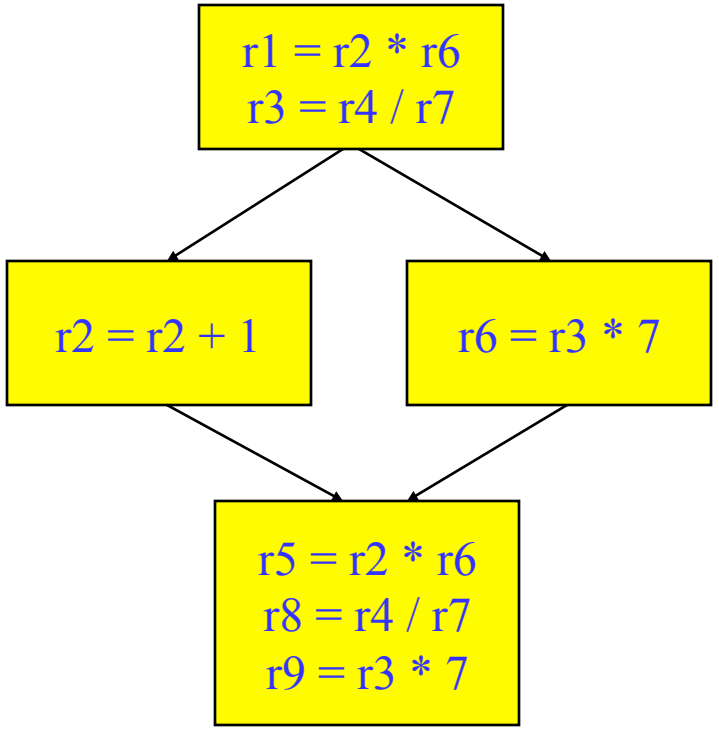
Forward Copy Propagation

- Forward propagation of the RHS of moves
  - r1 = r2
  - …
  - r4 = r1 + 1 → r4 = r2 + 1
- Benefits
  - Reduce chain of dependences
  - Eliminate the move
- Rules (ops X and Y)
  - X is a move
  - src1(X) is a register
  - Y consumes dest(X)
  - X.dest is an available def at Y
  - X.src1 is an available expr at Y



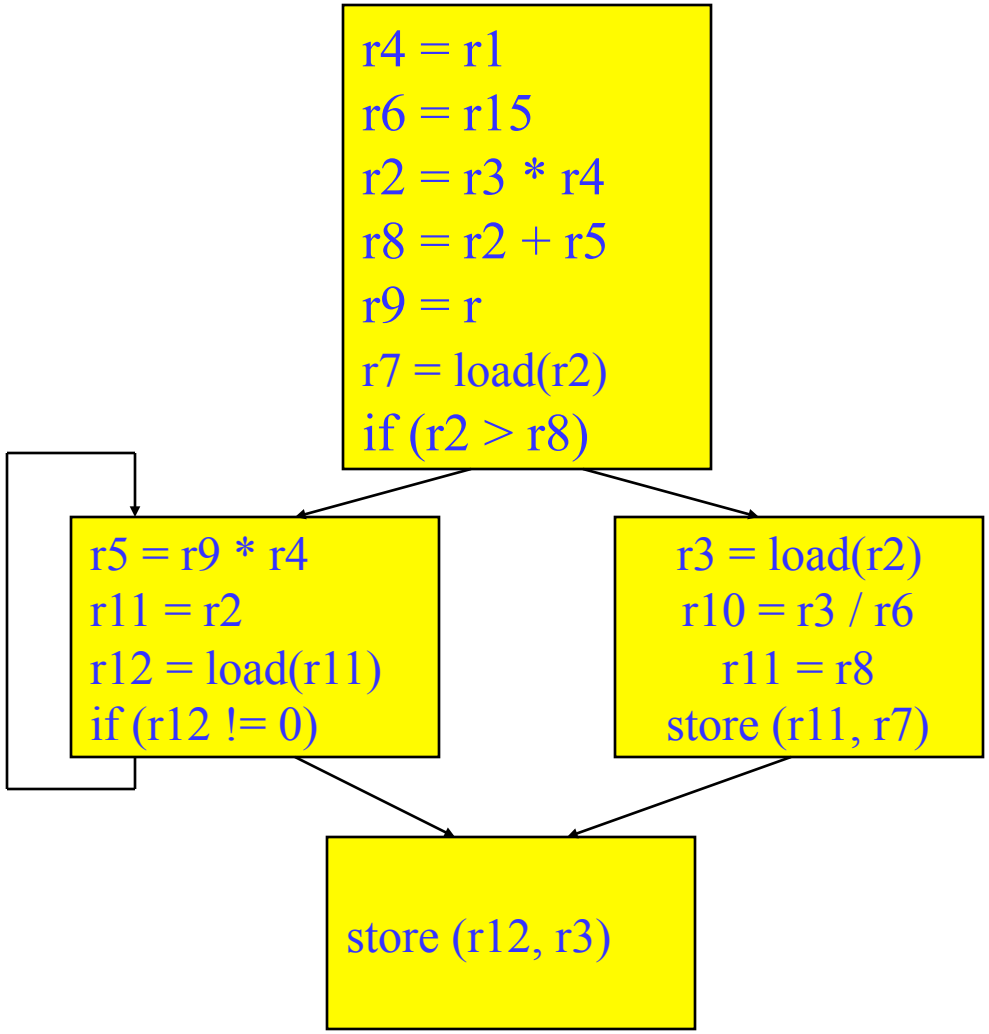
CSE – Common Subexpression Elimination

- Eliminate recomputation of an expression by reusing the previous result
  - r1 = r2 * r3
  - → r100 = r1
  - …
  - r4 = r2 * r3 → r4 = r100
- Benefits
  - Reduce work
  - Moves can get copy propagated
- Rules (ops X and Y)
  - X and Y have the same opcode
  - src(X) = src(Y), for all srcs
  - expr(X) is available at Y
  - if X is a load, then there is no store that may write to address(X) along any path between X and Y



if op is a load, call it redundant load elimination rather than CSE

Class Problem

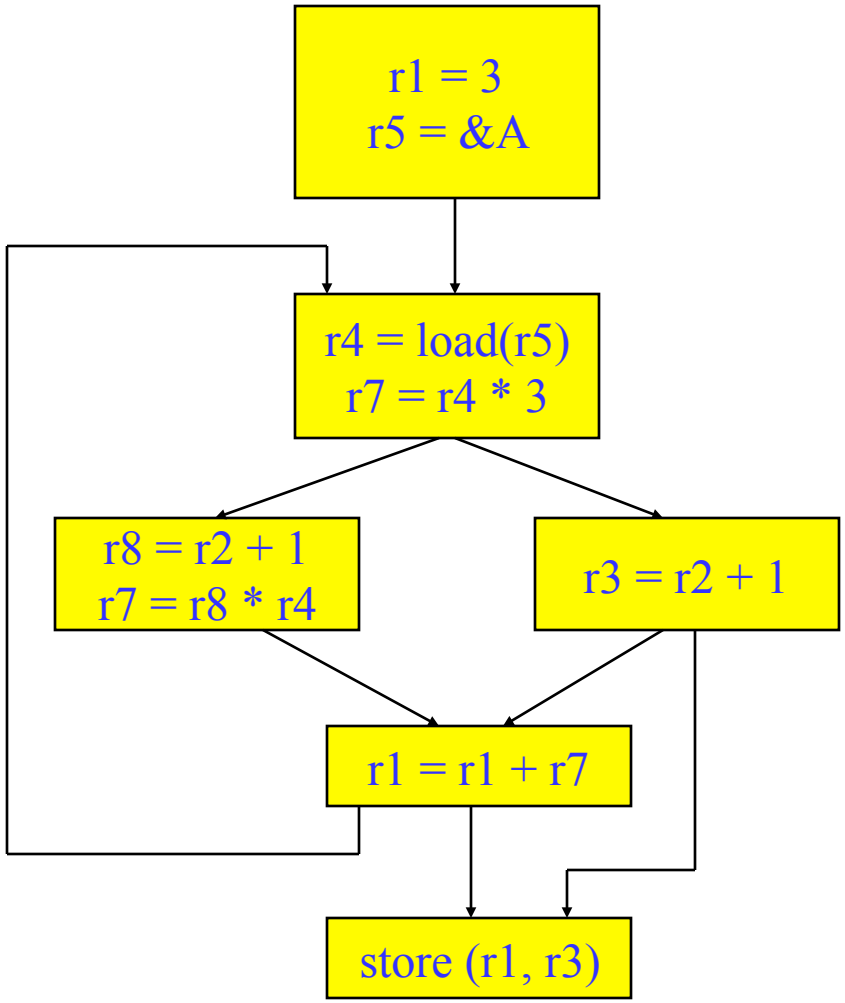


Optimize this applying

1. dead code elimination
2. forward copy propagation
3. CSE

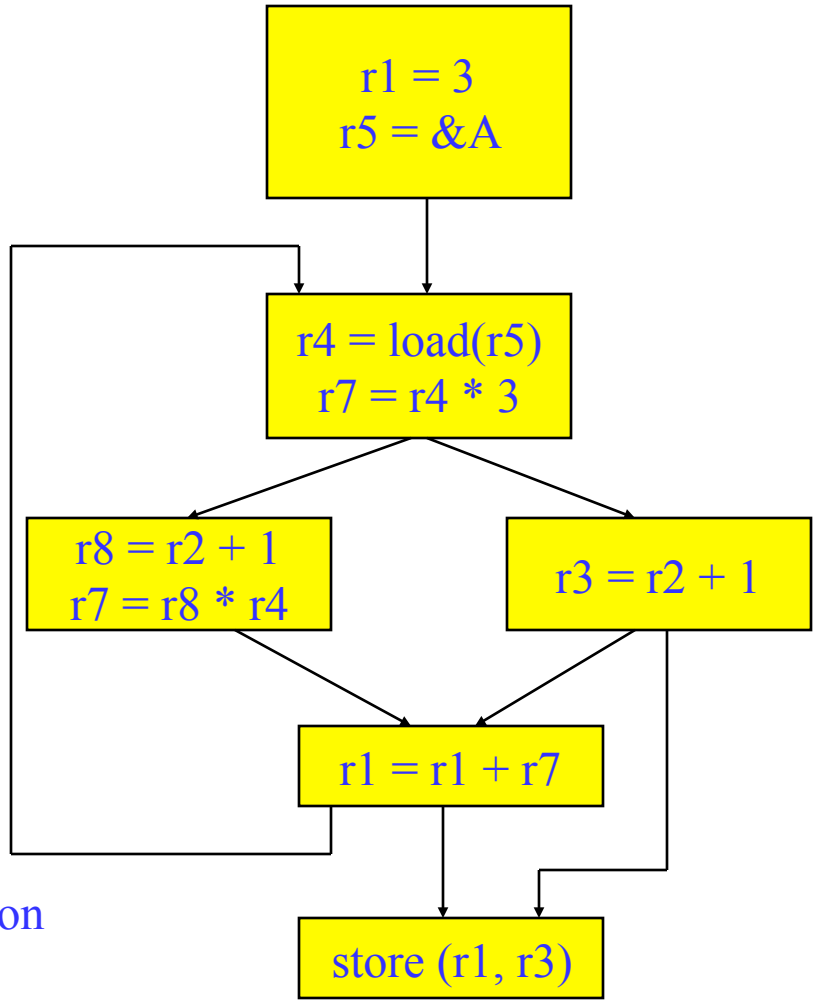
Loop Invariant Code Motion (LICM)

- Move operations whose source operands do not change within the loop to the loop preheader
  - » Execute them only 1x per invocation of the loop
  - » Be careful with memory operations!
  - » Be careful with ops not executed every iteration



LICM (2)

- Rules
  - X can be moved
  - src(X) not modified in loop body
  - X is the only op to modify dest(X)
  - for all uses of dest(X), X is in the available defs set
  - for all exit BB, if dest(X) is live on the exit edge, X is in the available defs set on the edge
  - if X not executed on every iteration, then X must provably not cause exceptions
  - if X is a load or store, then there are no writes to address(X) in loop



Homework 2 eliminates the last rule. You can also ignore the executed on every iteration rule for SpecLICM.

Global Variable Migration

- Assign a global variable temporarily to a register for the duration of the loop
  - » Load in preheader
  - » Store at exit points
- Rules
  - » X is a load or store
  - » address(X) not modified in the loop
  - » if X not executed on every iteration, then X must provably not cause an exception
  - » All memory ops in loop whose address can equal address(X) must always have the same address as X