EECS 583 – Class 7
Static Single Assignment Form

University of Michigan

September 22, 2021

Announcements & Reading Material

- HW2 out this past Monday
  - Spec and starting code are available on course webpage
  - Also check out piazza
- 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, Yay!
  - *Compilers: Principles, Techniques, and Tools*,
    A. Aho, R. Sethi, and J. Ullman, Addison-Wesley, 1988,
    9.9, 10.2, 10.3, 10.7 Edition 1; 8.5, 8.7, 9.1, 9.4, 9.5 Edition 2

From Last Time: What About All Path Problems?

- Up to this point
  - Any path problems (maybe relations)
    - Definition reaches along some path
    - Some sequence of branches in which def reaches
    - Lots of defs of the same variable may reach a point
  - Use of Union operator in meet function
- All-path: Definition guaranteed to reach
  - Regardless of sequence of branches taken, def reaches
  - Can always count on this
  - Only 1 def can be guaranteed to reach
  - Availability (as opposed to reaching)
    - Available definitions
    - Available expressions (could also have reaching expressions, but not that useful)

Available Definition Analysis (Adefs)

- A definition d is available at a point p if along all paths from d to p, d is not killed
- Remember, a definition of a variable is killed between 2 points when there is another definition of that variable along the path
  - r1 = r2 + r3 kills previous definitions of r1
- Algorithm
  - Forward dataflow analysis as propagation occurs from defs downwards
  - Use the Intersect function as the meet operator to guarantee the all-path requirement
  - GEN/KILL/IN/OUT similar to reaching defs
    - Initialization of IN/OUT is the tricky part

Compute GEN/KILL Sets for each BB (Adefs)

Exactly the same as reaching defs !!!

```
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
        for each destination operand of op, dest, do
            G = op
            K = {all ops which define dest – op}
            GEN(X) = G + (GEN(X) – K)
            KILL(X) = K + (KILL(X) – G)
        endfor
    endfor
endwhile
```

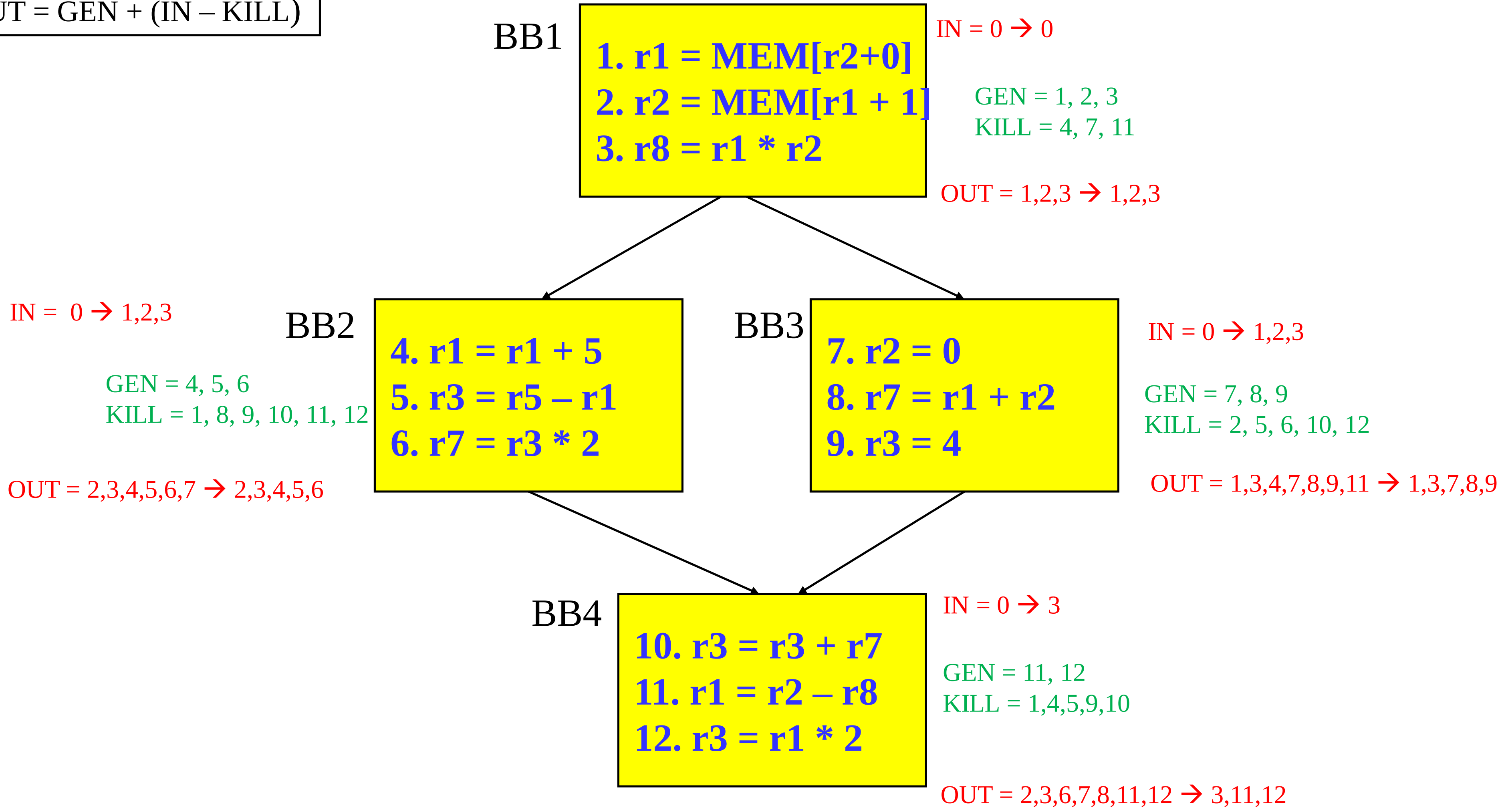
Compute IN/OUT Sets for all BBs (Adefs)

```
U = universal set of all operations in the Procedure
IN(0) = 0
OUT(0) = GEN(0)
for each basic block in procedure, W, (W != 0), do
    IN(W) = 0
    OUT(W) = U – KILL(W)

change = 1
while (change) do
    change = 0
    for each basic block in procedure, X, do
        old_OUT = OUT(X)
        IN(X) = Intersect(OUT(Y)) for all predecessors Y of X
        OUT(X) = GEN(X) + (IN(X) – KILL(X))
        if (old_OUT != OUT(X)) then
            change = 1
        endif
    endfor
endwhile
```

Example Adef Calculation

IN = Intersect(OUT(preds))
OUT = GEN + (IN – KILL)



BB1

1. r1 = MEM[r2+0]
2. r2 = MEM[r1 + 1]
3. r8 = r1 * r2

IN = 0 → 0

GEN = 1, 2, 3
KILL = 4, 7, 11

OUT = 1,2,3 → 1,2,3

BB2

4. r1 = r1 + 5
5. r3 = r5 – r1
6. r7 = r3 * 2

IN = 0 → 1,2,3

GEN = 4, 5, 6
KILL = 1, 8, 9, 10, 11, 12

OUT = 2,3,4,5,6,7 → 2,3,4,5,6

BB3

7. r2 = 0
8. r7 = r1 + r2
9. r3 = 4

IN = 0 → 1,2,3

GEN = 7, 8, 9
KILL = 2, 5, 6, 10, 12

OUT = 1,3,4,7,8,9,11 → 1,3,7,8,9

BB4

10. r3 = r3 + r7
11. r1 = r2 – r8
12. r3 = r1 * 2

IN = 0 → 3

GEN = 11, 12
KILL = 1,4,5,9,10

OUT = 2,3,6,7,8,11,12 → 3,11,12

Available Expression Analysis (Aexprs)

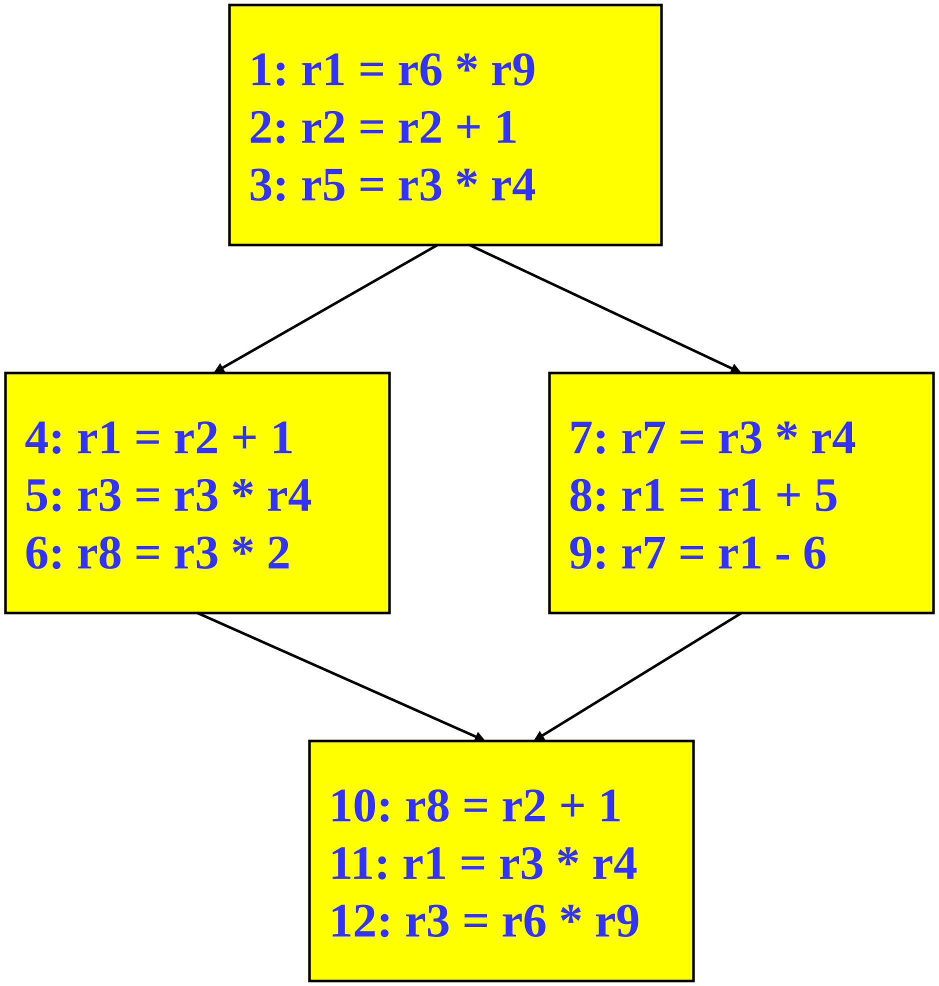
- An expression is a RHS of an operation
  - r2 = r3 + r4, r3+r4 is an expression
- An expression e is available at a point p if along all paths from e to p, e is not killed
- An expression is killed between 2 points when one of its source operands are redefined
  - r1 = r2 + r3 kills all expressions involving r1
- Algorithm
  - Forward dataflow analysis as propagation occurs from defs downwards
  - Use the Intersect function as the meet operator to guarantee the all-path requirement
  - Looks exactly like adefs, except GEN/KILL/IN/OUT are the RHS's of operations rather than the LHS's

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
```

Homework Problem - Aexprs Calculation
Answer on the Next Slide



```
1: r1 = r6 * r9
2: r2 = r2 + 1
3: r5 = r3 * r4
```

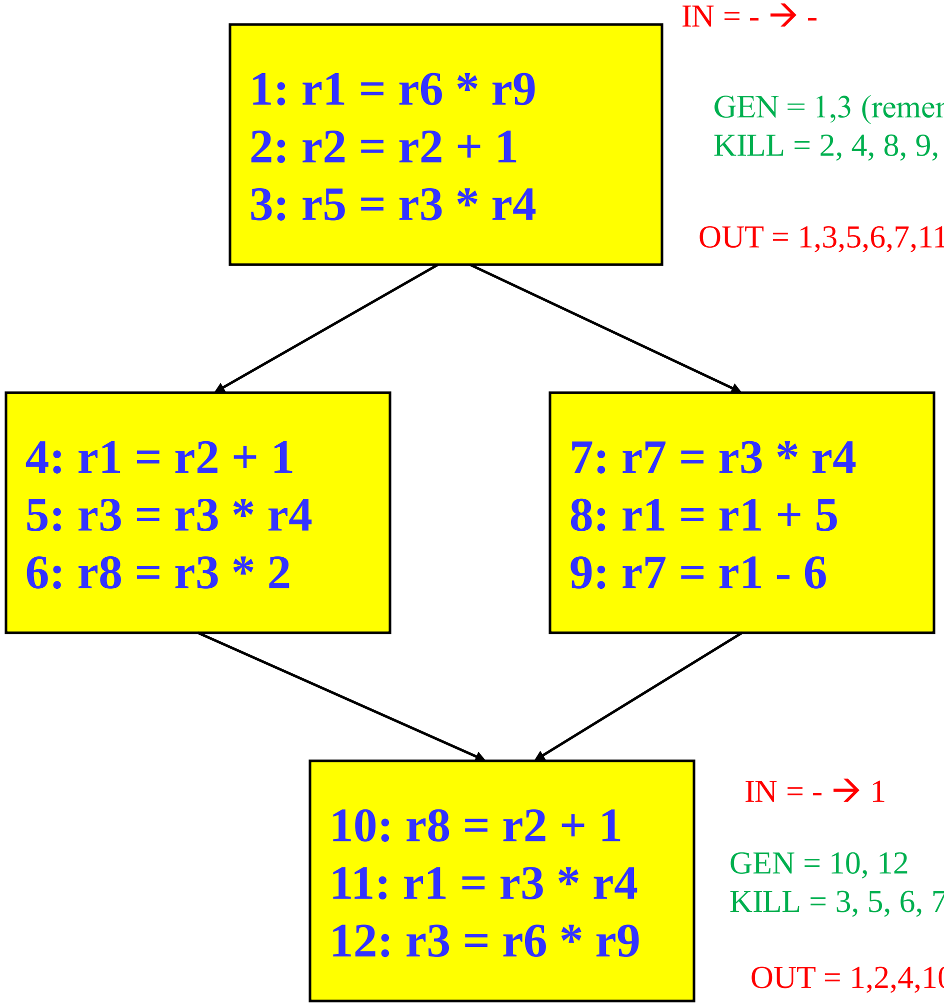
```
4: r1 = r2 + 1
5: r3 = r3 * r4
6: r8 = r3 * 2
```

```
7: r7 = r3 * r4
8: r1 = r1 + 5
9: r7 = r1 - 6
```

```
10: r8 = r2 + 1
11: r1 = r3 * r4
12: r3 = r6 * r9
```

Homework Problem - Answer

IN/OUT sets
A → B
A = initial state
B = after first iteration



```
1: r1 = r6 * r9
2: r2 = r2 + 1
3: r5 = r3 * r4
```

IN = - → -

GEN = 1,3 (remember {1, 3} means {“r6*r9”, “r3*r4”})
KILL = 2, 4, 8, 9, 10

OUT = 1,3,5,6,7,11,12 → 1,3

```
4: r1 = r2 + 1
5: r3 = r3 * r4
6: r8 = r3 * 2
```

IN = - → 1,3

GEN = 4, 6
KILL = 3, 5, 7, 8, 9, 11

OUT = 1,2,4,6,10,12 → 1,4,6

```
7: r7 = r3 * r4
8: r1 = r1 + 5
9: r7 = r1 - 6
```

IN = - → 1,3

GEN = 7, 9
KILL = 8

OUT = 1,2,3,4,5,6,7,9,10,11,12 → 1,3,7,9

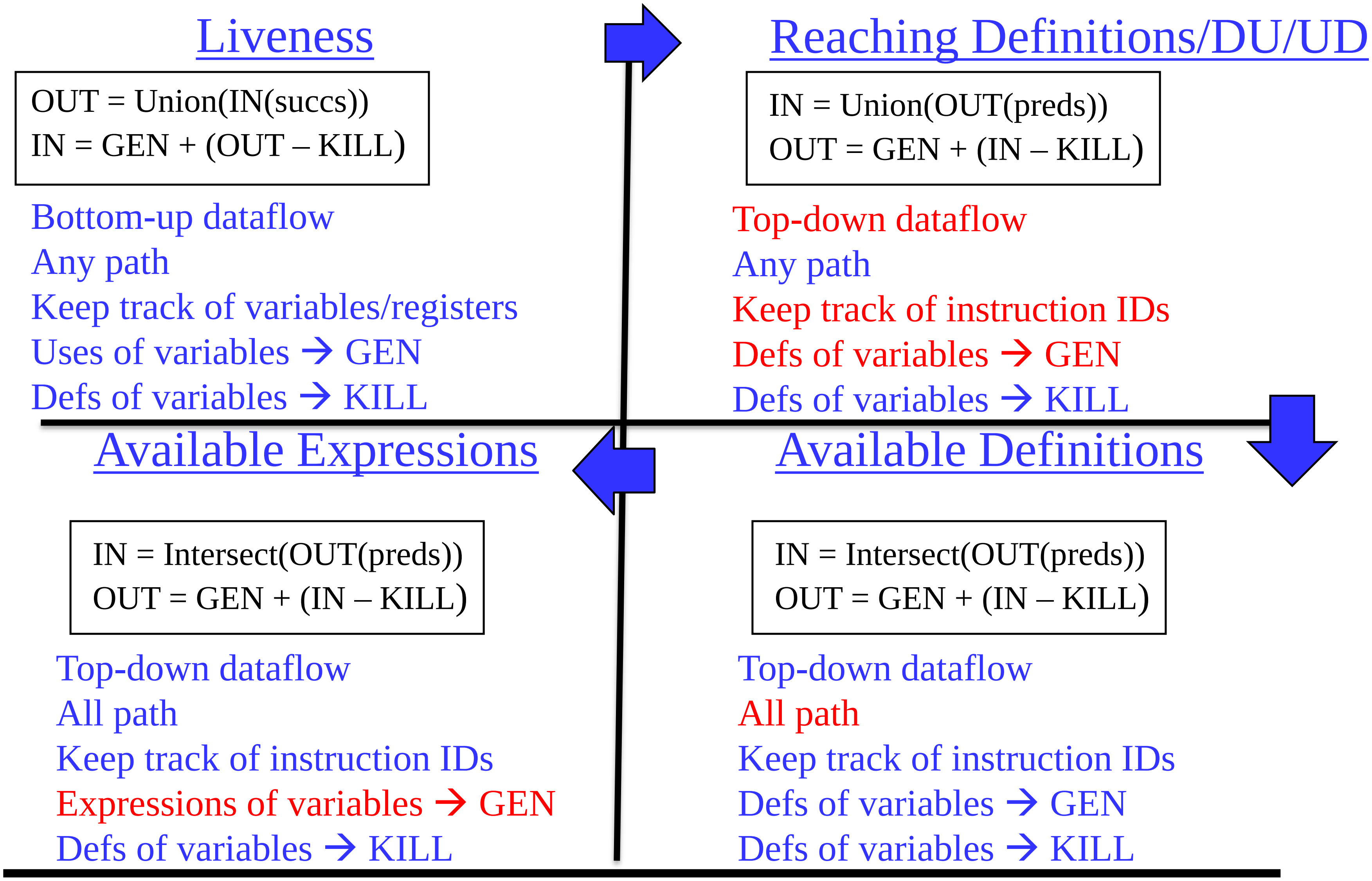
```
10: r8 = r2 + 1
11: r1 = r3 * r4
12: r3 = r6 * r9
```

IN = - → 1

GEN = 10, 12
KILL = 3, 5, 6, 7, 8, 9, 11

OUT = 1,2,4,10,12 → 1,10,12

Dataflow Summary Analyses in 1 Slide



Liveness

OUT = Union(IN(succs))
IN = GEN + (OUT – KILL)

Bottom-up dataflow
Any path
Keep track of variables/registers
Uses of variables → GEN
Defs of variables → KILL

Reaching Definitions/DU/UD

IN = Union(OUT(preds))
OUT = GEN + (IN – KILL)

Top-down dataflow
Any path
Keep track of instruction IDs
Defs of variables → GEN
Defs of variables → KILL

Available Expressions

IN = Intersect(OUT(preds))
OUT = GEN + (IN – KILL)

Top-down dataflow
All path
Keep track of instruction IDs
Expressions of variables → GEN
Defs of variables → KILL

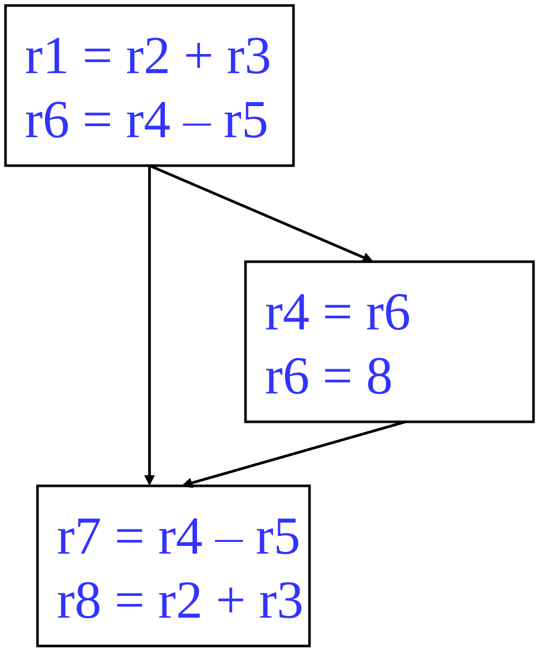
Available Definitions

IN = Intersect(OUT(preds))
OUT = GEN + (IN – KILL)

Top-down dataflow
All path
Keep track of instruction IDs
Defs of variables → GEN
Defs of variables → KILL

Static Single Assignment (SSA) Form

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



r1 = r2 + r3
r6 = r4 – r5

r4 = r6
r6 = 8

r7 = r4 – r5
r8 = r2 + r3

- 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)
```

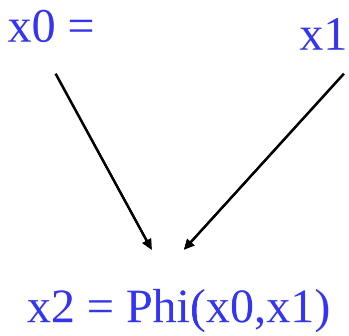
```
i0 = 0
do {
    i1 = Phi(i0, i2)
    i2 = i1 + 1
}
while (i2 < 50)
```

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



x0 =

x1 =

x2 = Phi(x0,x1)

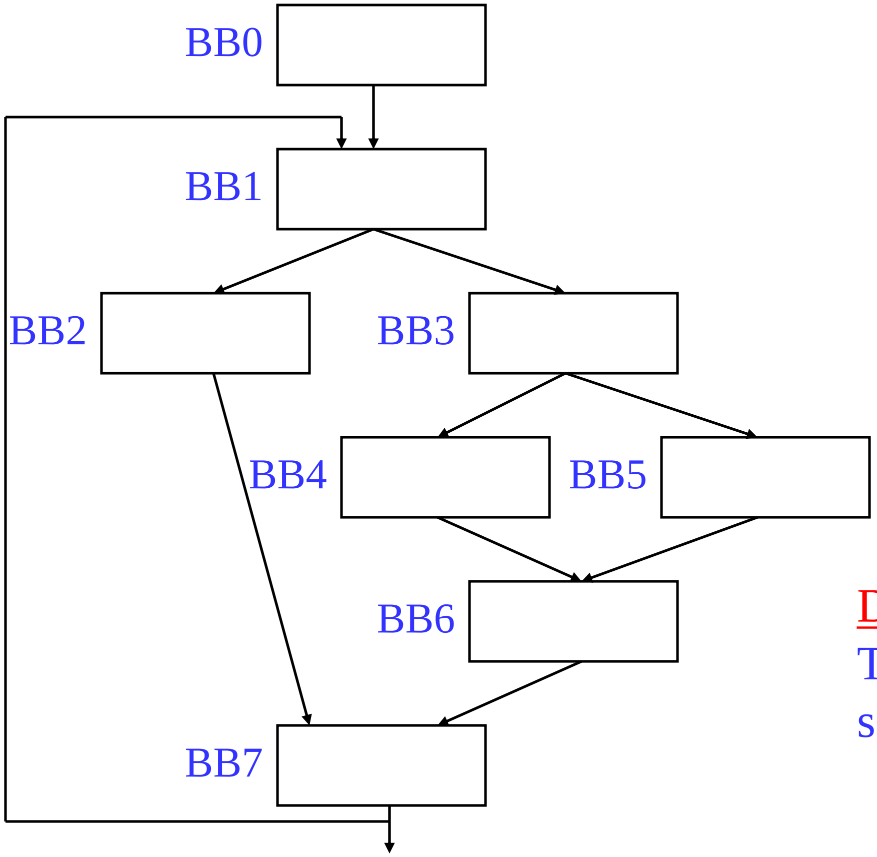
- Phi nodes are required when 2 non-null paths X→Z and Y→Z converge at node Z, and nodes X and Y contain assignments to V

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

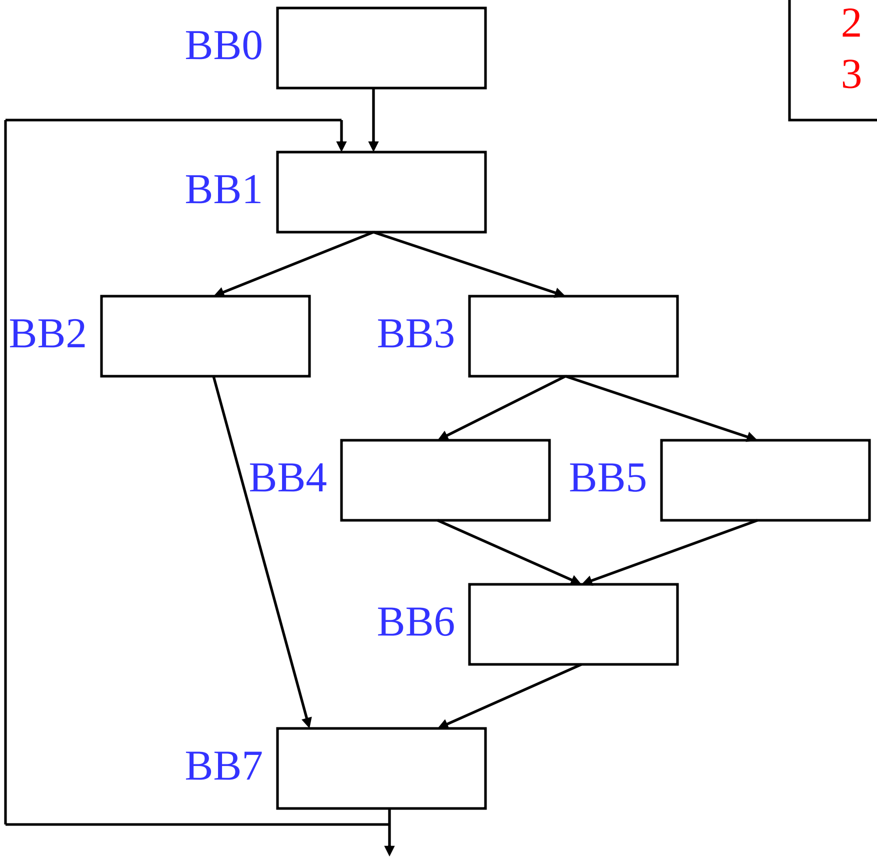
* X dominates a predecessor of Y, but

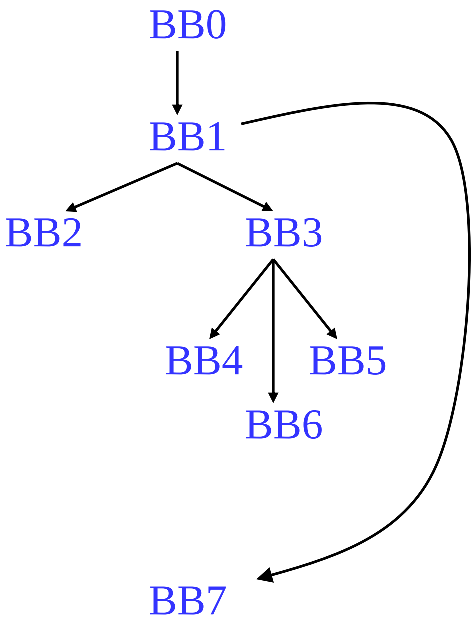
* X does not strictly dominate Y

Recall: Dominator Tree

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

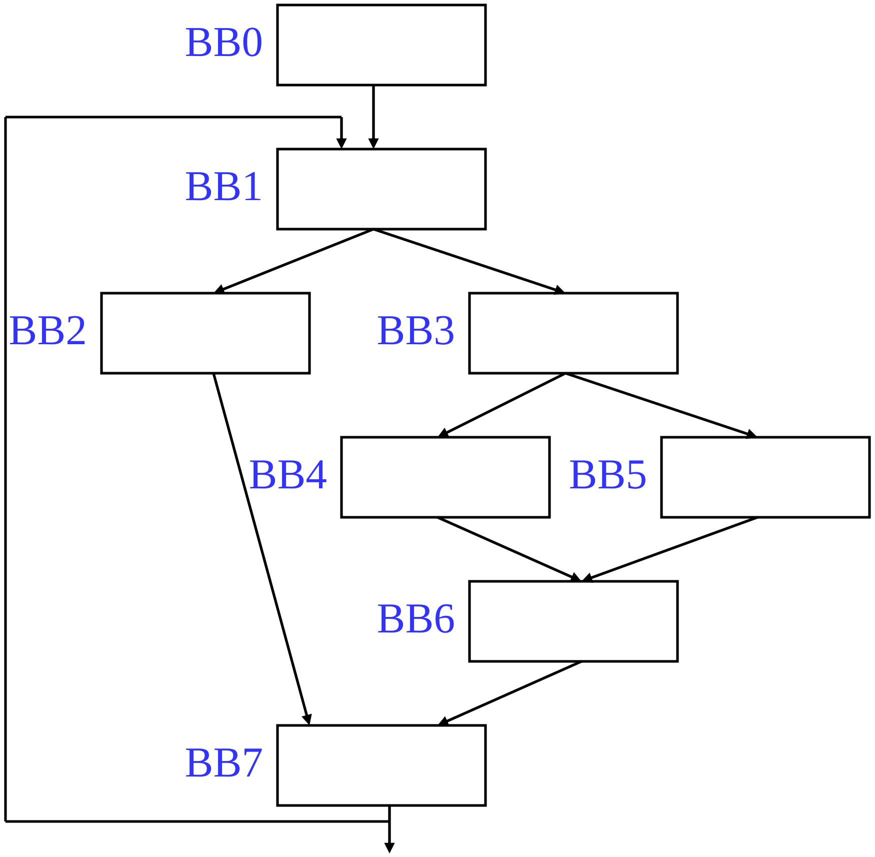
BB	DOM	BB	DOM
0	0	4	0,1,3,4
1	0,1	5	0,1,3,5
2	0,1,2	6	0,1,3,6
3	0,1,3	7	0,1,7

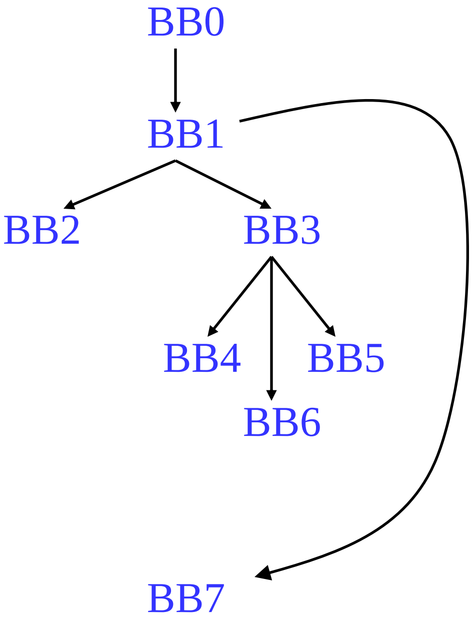




Dom tree

Computing Dominance Frontiers

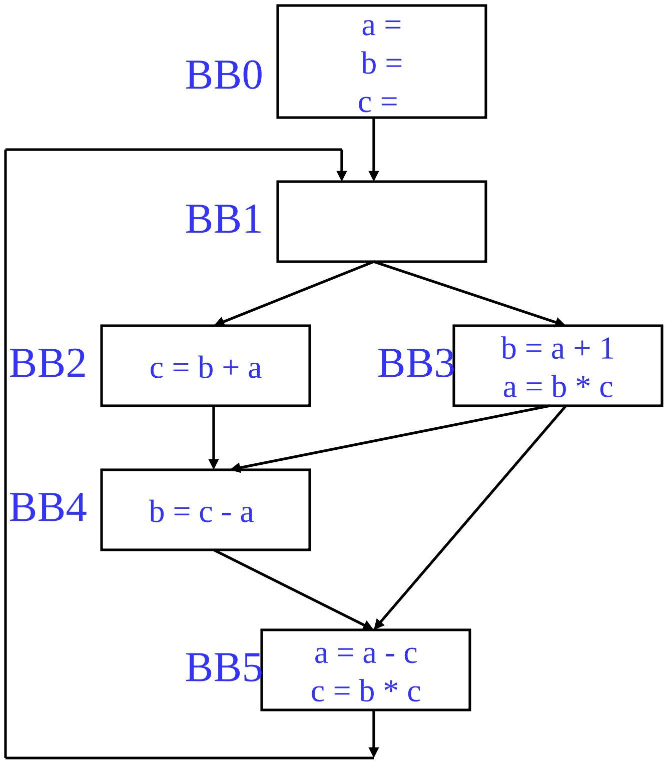




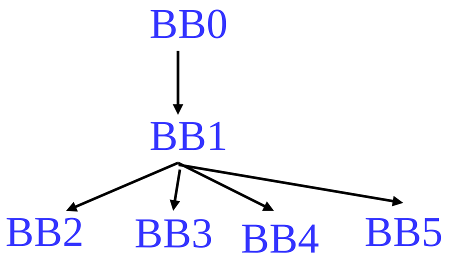
BB	DF
0	
1	
2	
3	
4	
5	
6	
7	

```
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)
```

Homework Problem – Compute DF for each BB Answer on Slide 24



Dominator Tree



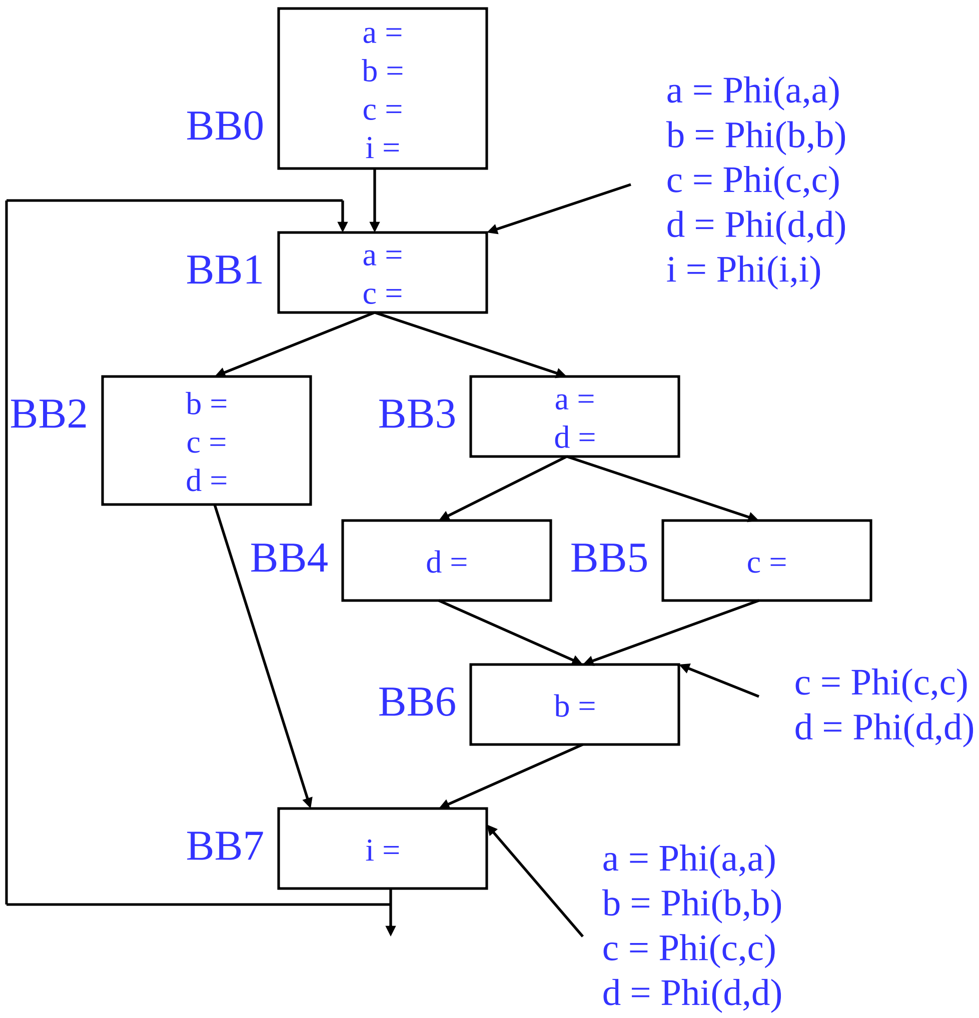
```
For each join point X in the CFG
    For each predecessor, Y, of X in the CFG
        Run up to the IDOM(X) in the dominator tree,
        adding X to DF(N) for each N between Y and
        IDOM(X) (or X, whichever is encountered first)
```

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



BB0

a =
b =
c =
i =

a = Phi(a,a)
b = Phi(b,b)
c = Phi(c,c)
d = Phi(d,d)
i = Phi(i,i)

BB1

a =
c =

BB2

b =
c =
d =

BB3

a =
d =

BB4

d =

BB5

c =

BB6

b =

c = Phi(c,c)
d = Phi(d,d)

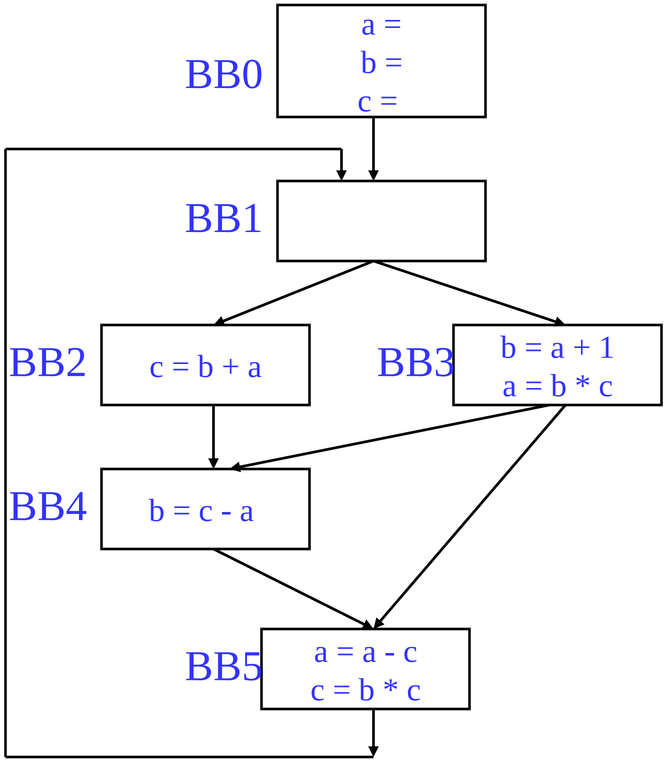
BB7

i =

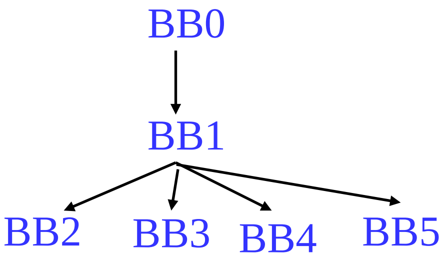
a = Phi(a,a)
b = Phi(b,b)
c = Phi(c,c)
d = Phi(d,d)

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

Homework Problem – Insert Phi Nodes
Answer on Slide 36



Dominator tree



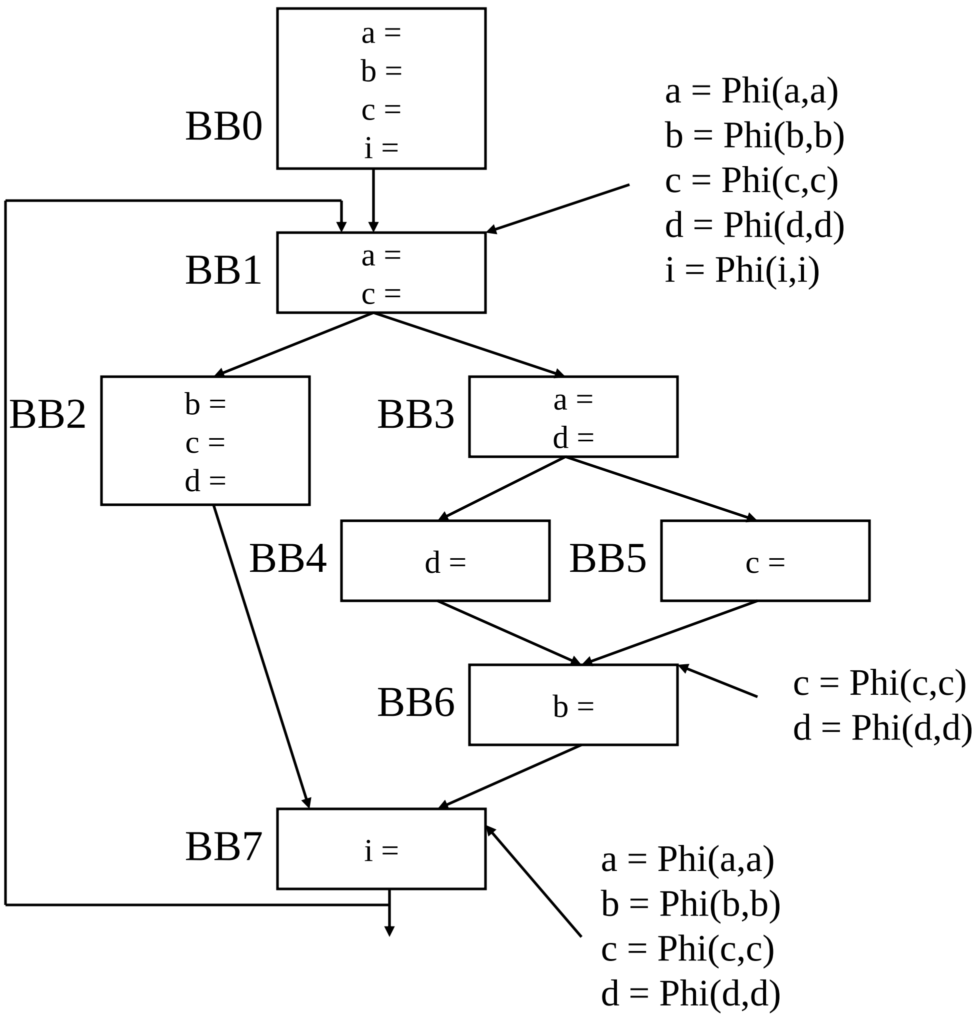
Dominance frontier

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

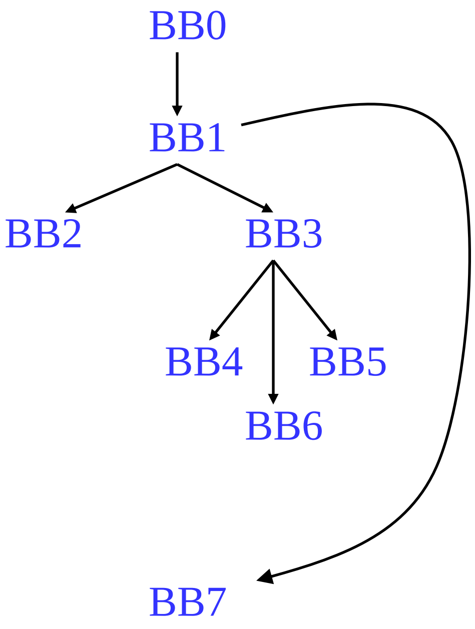
Renaming – Example (Initial State)



a = Phi(a,a)
b = Phi(b,b)
c = Phi(c,c)
d = Phi(d,d)
i = Phi(i,i)

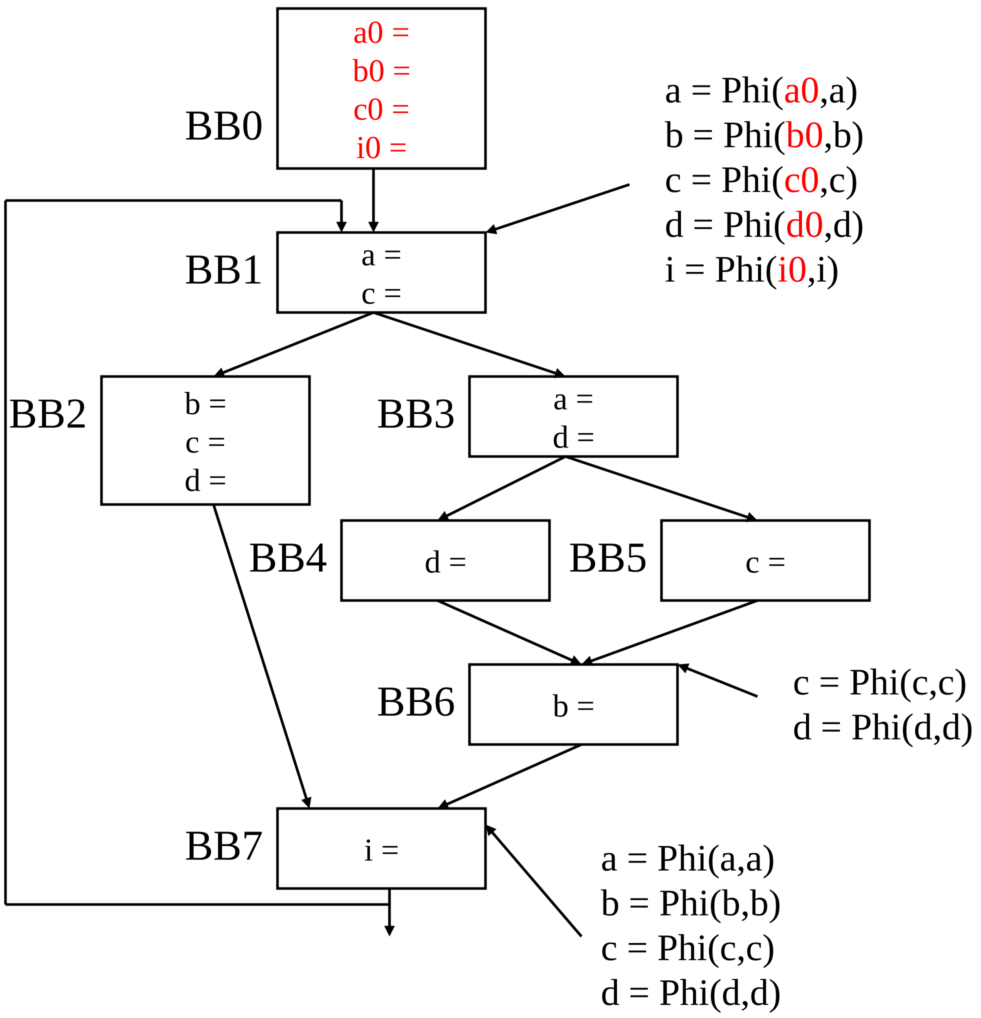
c = Phi(c,c)
d = Phi(d,d)

a = Phi(a,a)
b = Phi(b,b)
c = Phi(c,c)
d = Phi(d,d)



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

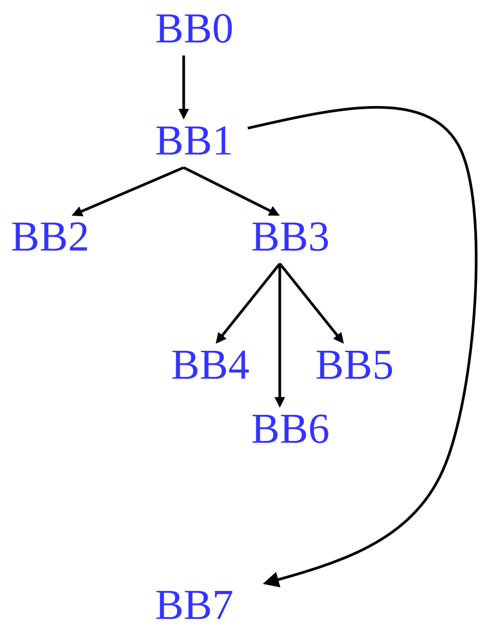
Renaming – Example (After BB0)



a = Phi(a0,a)
b = Phi(b0,b)
c = Phi(c0,c)
d = Phi(d0,d)
i = Phi(i0,i)

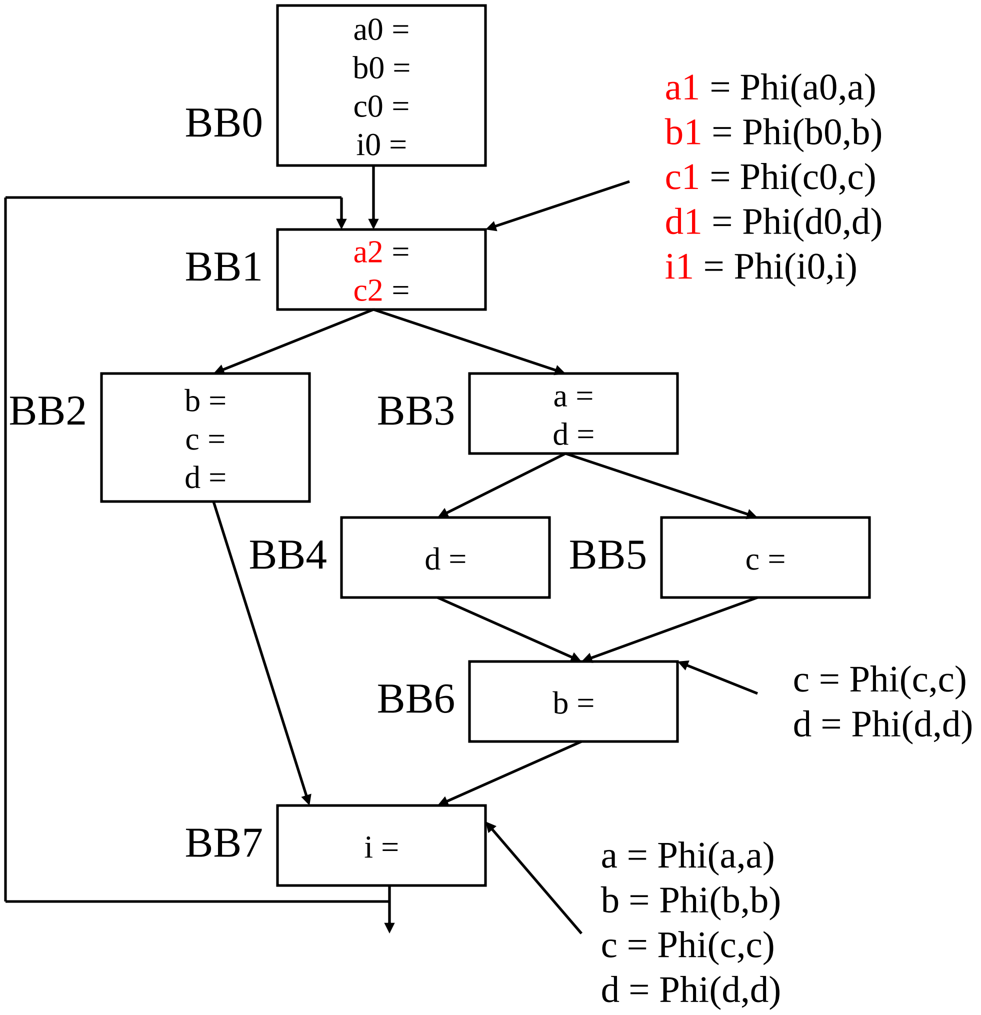
c = Phi(c,c)
d = Phi(d,d)

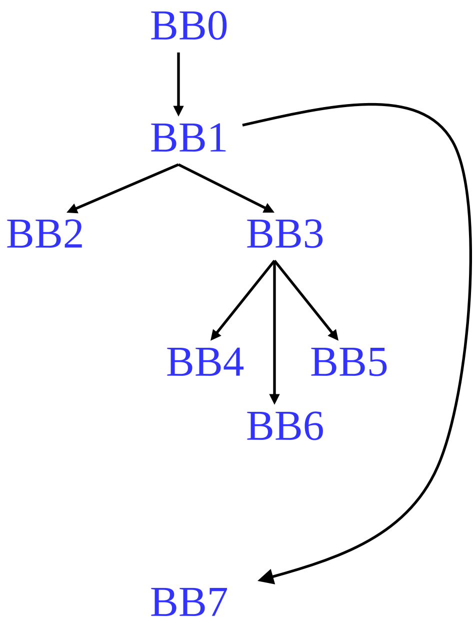
a = Phi(a,a)
b = Phi(b,b)
c = Phi(c,c)
d = Phi(d,d)



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

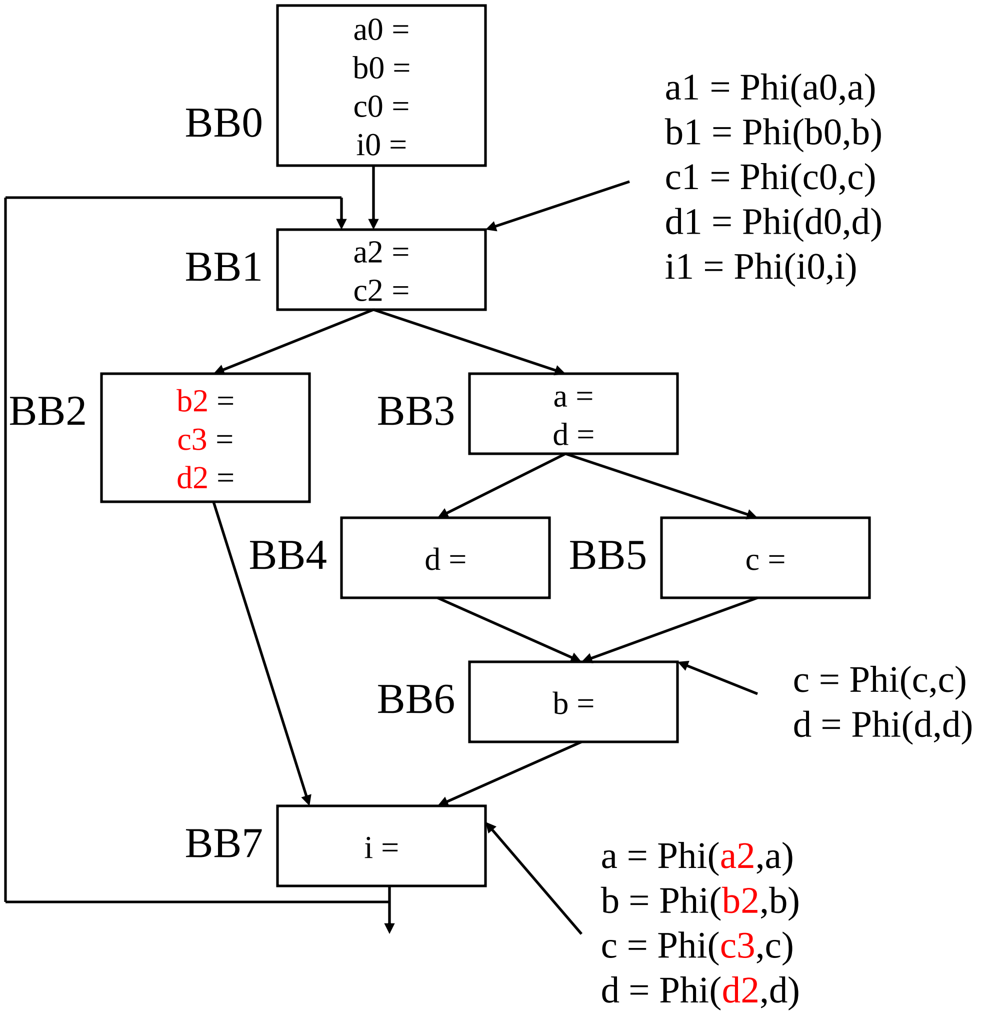
Renaming – Example (After BB1)





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)



BB0

a0 =
b0 =
c0 =
i0 =

BB1

a2 =
c2 =

a1 = Phi(a0,a)
b1 = Phi(b0,b)
c1 = Phi(c0,c)
d1 = Phi(d0,d)
i1 = Phi(i0,i)

BB2

b2 =
c3 =
d2 =

BB3

a =
d =

BB4

d =

BB5

c =

BB6

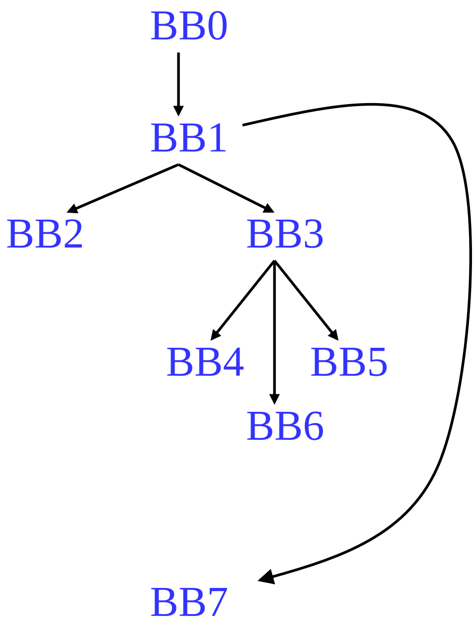
b =

c = Phi(c,c)
d = Phi(d,d)

BB7

i =

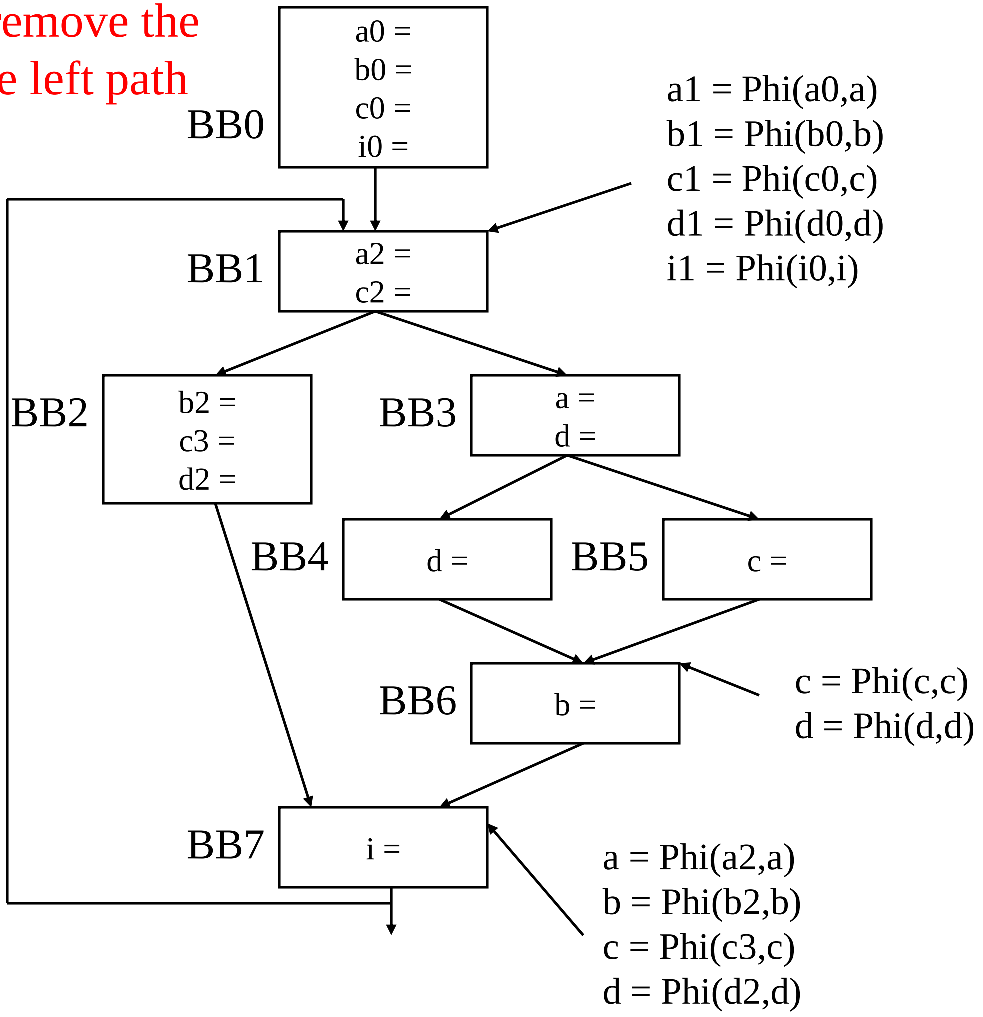
a = Phi(a2,a)
b = Phi(b2,b)
c = Phi(c3,c)
d = Phi(d2,d)



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



BB0
a0 =
b0 =
c0 =
i0 =

BB1
a2 =
c2 =

a1 = Phi(a0,a)
b1 = Phi(b0,b)
c1 = Phi(c0,c)
d1 = Phi(d0,d)
i1 = Phi(i0,i)

BB2
b2 =
c3 =
d2 =

BB3
a =
d =

BB4
d =

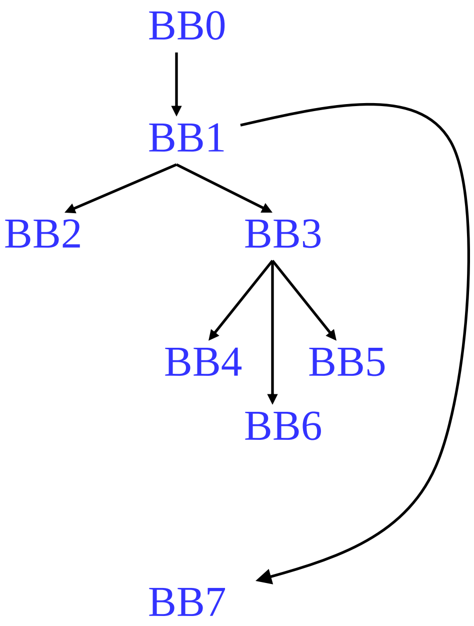
BB5
c =

BB6
b =

c = Phi(c,c)
d = Phi(d,d)

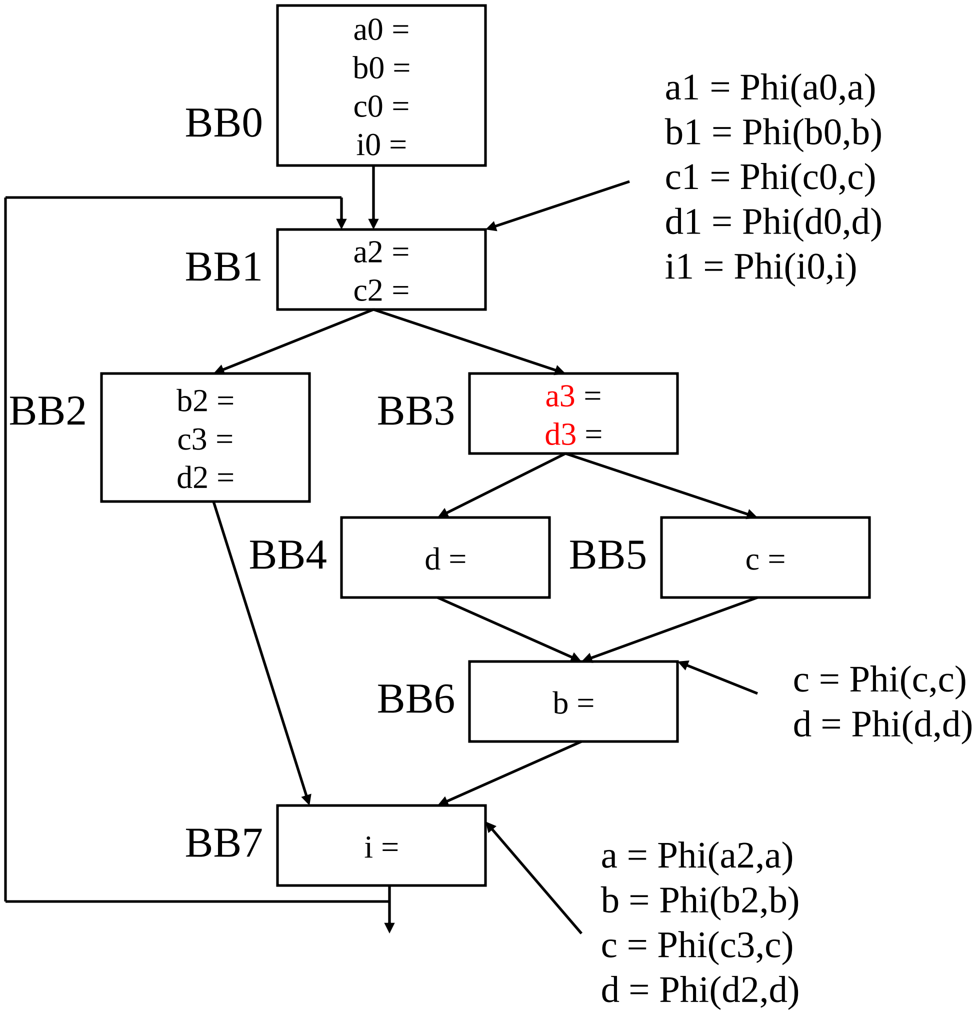
BB7
i =

a = Phi(a2,a)
b = Phi(b2,b)
c = Phi(c3,c)
d = Phi(d2,d)



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)



BB0

a0 =
b0 =
c0 =
i0 =

BB1

a2 =
c2 =

a1 = Phi(a0,a)
b1 = Phi(b0,b)
c1 = Phi(c0,c)
d1 = Phi(d0,d)
i1 = Phi(i0,i)

BB2

b2 =
c3 =
d2 =

BB3

a3 =
d3 =

BB4

d =

BB5

c =

BB6

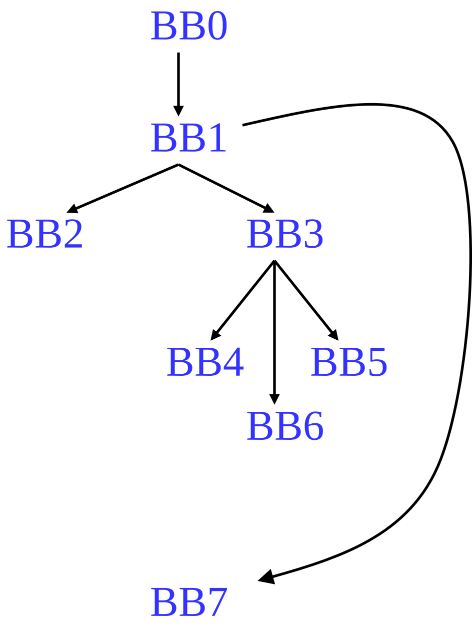
b =

c = Phi(c,c)
d = Phi(d,d)

BB7

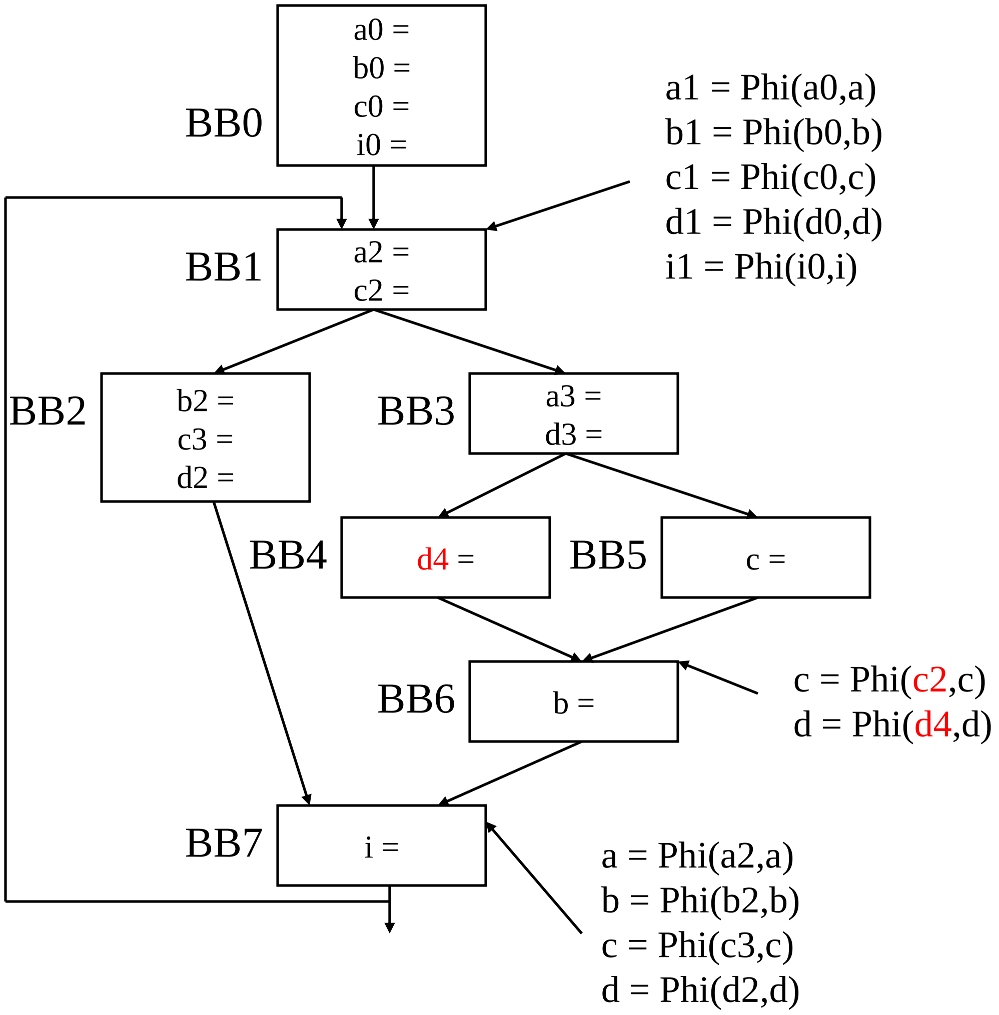
i =

a = Phi(a2,a)
b = Phi(b2,b)
c = Phi(c3,c)
d = Phi(d2,d)



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)



BB0

a0 =
b0 =
c0 =
i0 =

BB1

a2 =
c2 =

a1 = Phi(a0,a)
b1 = Phi(b0,b)
c1 = Phi(c0,c)
d1 = Phi(d0,d)
i1 = Phi(i0,i)

BB2

b2 =
c3 =
d2 =

BB3

a3 =
d3 =

BB4

d4 =

BB5

c =

BB6

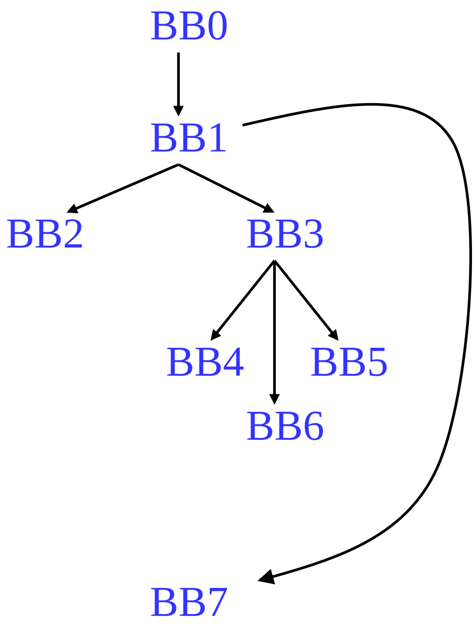
b =

c = Phi(c2,c)
d = Phi(d4,d)

BB7

i =

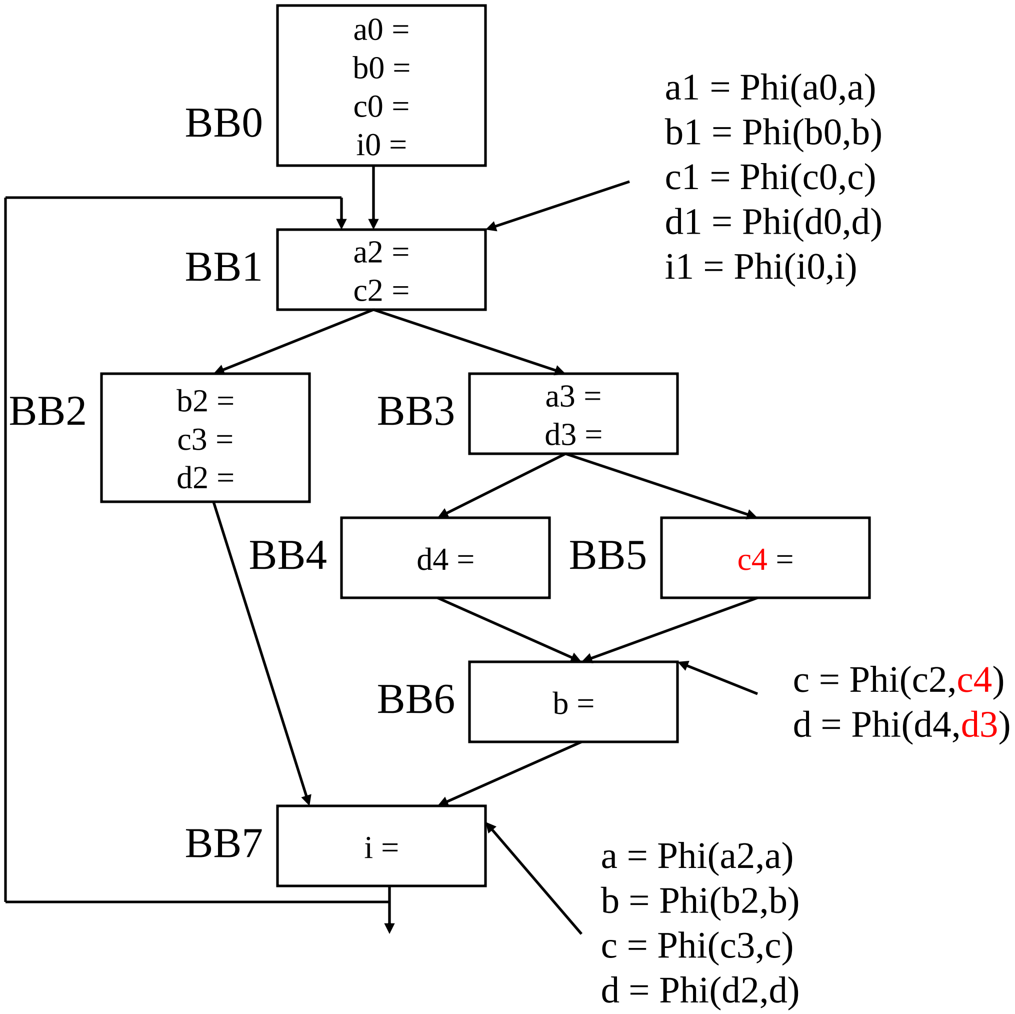
a = Phi(a2,a)
b = Phi(b2,b)
c = Phi(c3,c)
d = Phi(d2,d)



BB0
BB1
BB2
BB3
BB4
BB5
BB6
BB7

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)



BB0

a0 =
b0 =
c0 =
i0 =

BB1

a2 =
c2 =

a1 = Phi(a0,a)
b1 = Phi(b0,b)
c1 = Phi(c0,c)
d1 = Phi(d0,d)
i1 = Phi(i0,i)

BB2

b2 =
c3 =
d2 =

BB3

a3 =
d3 =

BB4

d4 =

BB5

c4 =

BB6

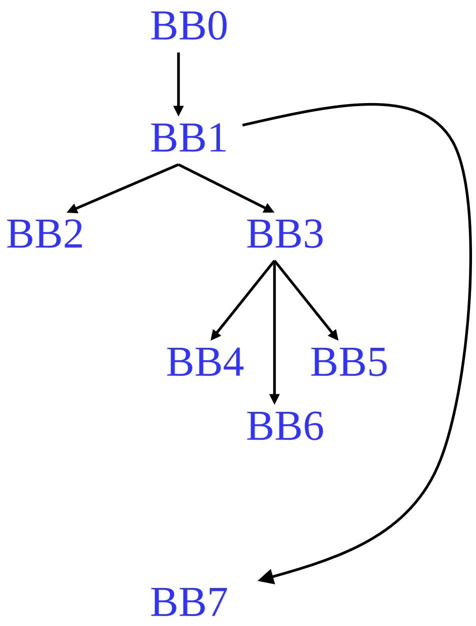
b =

c = Phi(c2,c4)
d = Phi(d4,d3)

BB7

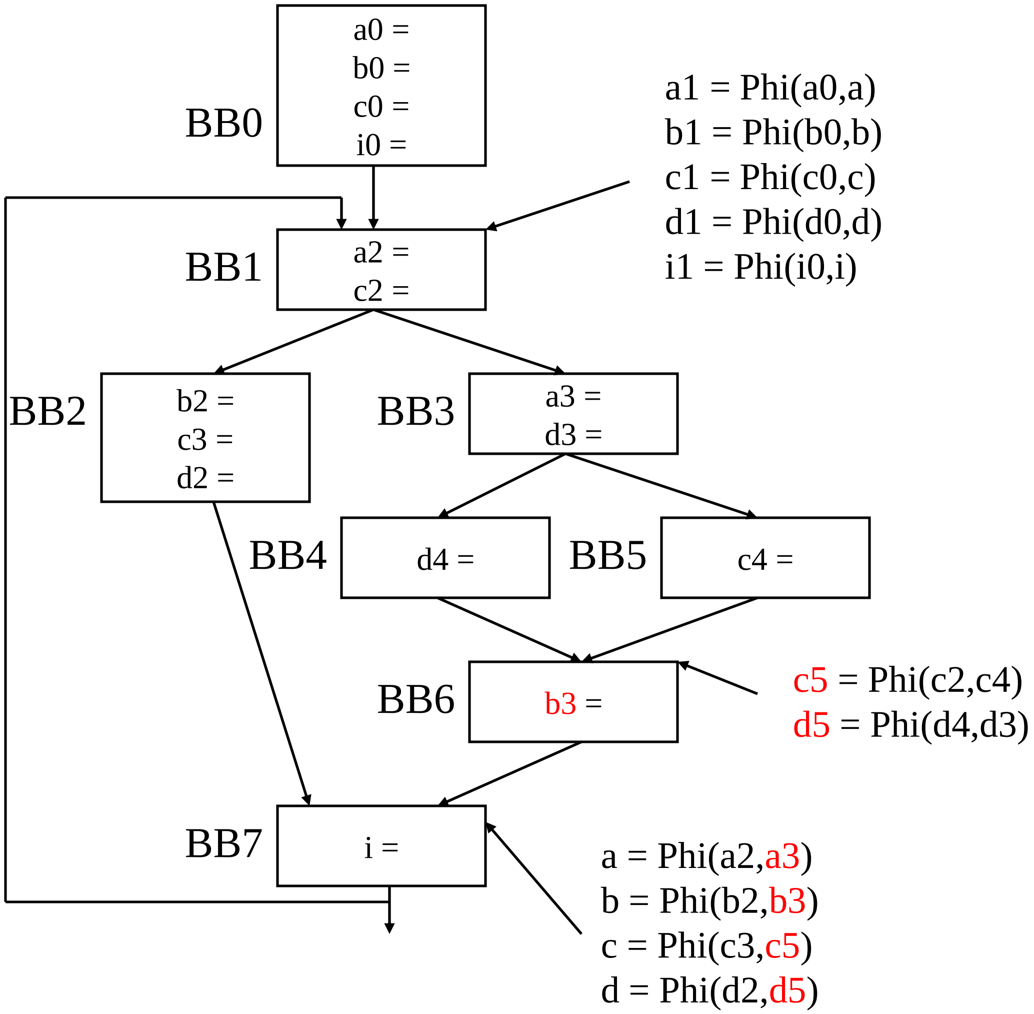
i =

a = Phi(a2,a)
b = Phi(b2,b)
c = Phi(c3,c)
d = Phi(d2,d)



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)



a0 =
b0 =
c0 =
i0 =

a2 =
c2 =

b2 =
c3 =
d2 =

a3 =
d3 =

d4 =

c4 =

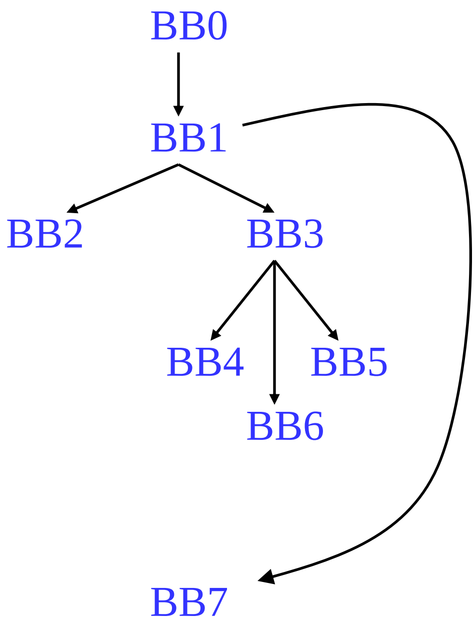
b3 =

i =

a1 = Phi(a0,a)
b1 = Phi(b0,b)
c1 = Phi(c0,c)
d1 = Phi(d0,d)
i1 = Phi(i0,i)

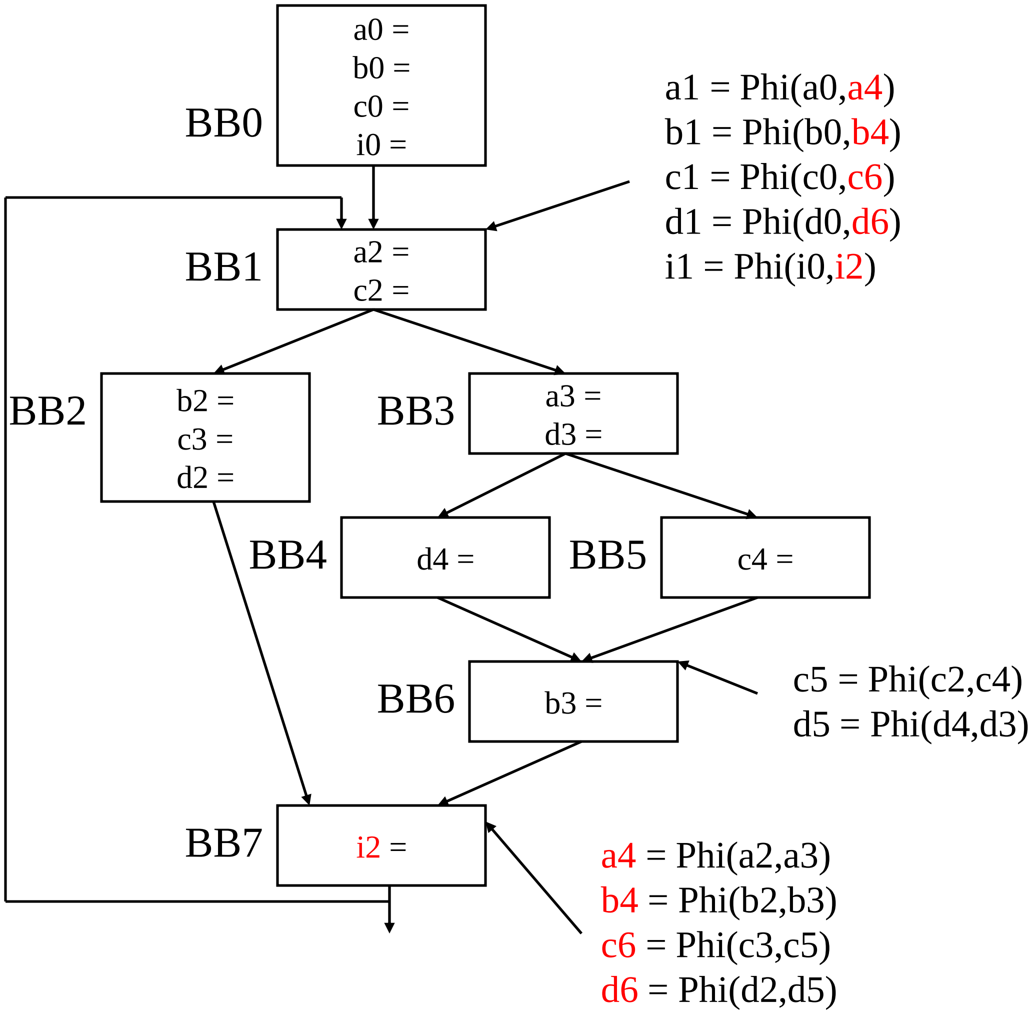
c5 = Phi(c2,c4)
d5 = Phi(d4,d3)

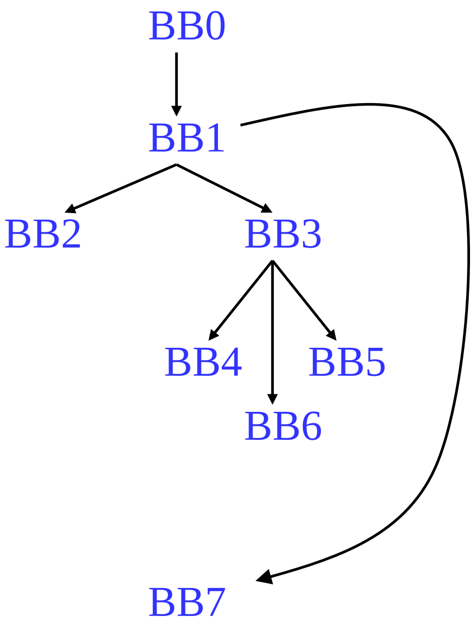
a = Phi(a2,a3)
b = Phi(b2,b3)
c = Phi(c3,c5)
d = Phi(d2,d5)



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)

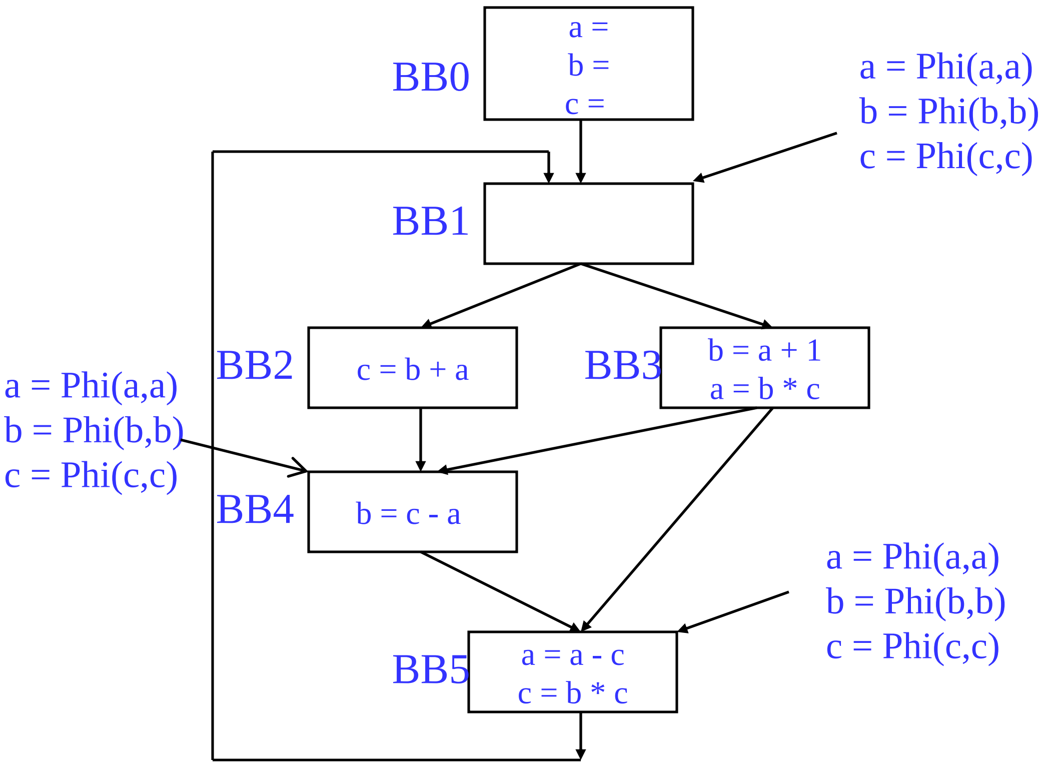




var:	a	b	c	d	i
ctr:	5	5	7	7	3
stk:	a0	b0	c0	d0	i0
	a1	b1	c1	d1	i1
	a2	b4	c2	d6	i2
	a4		c6		

Fin!

Homework Problem – Rename the Variables



BB0

a =
b =
c =

BB1

a = Phi(a,a)
b = Phi(b,b)
c = Phi(c,c)

BB2

c = b + a

BB3

b = a + 1
a = b * c

a = Phi(a,a)
b = Phi(b,b)
c = Phi(c,c)

BB4

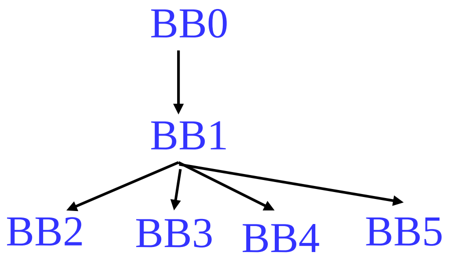
b = c - a

BB5

a = Phi(a,a)
b = Phi(b,b)
c = Phi(c,c)

a = a - c
c = b * c

Dominator tree



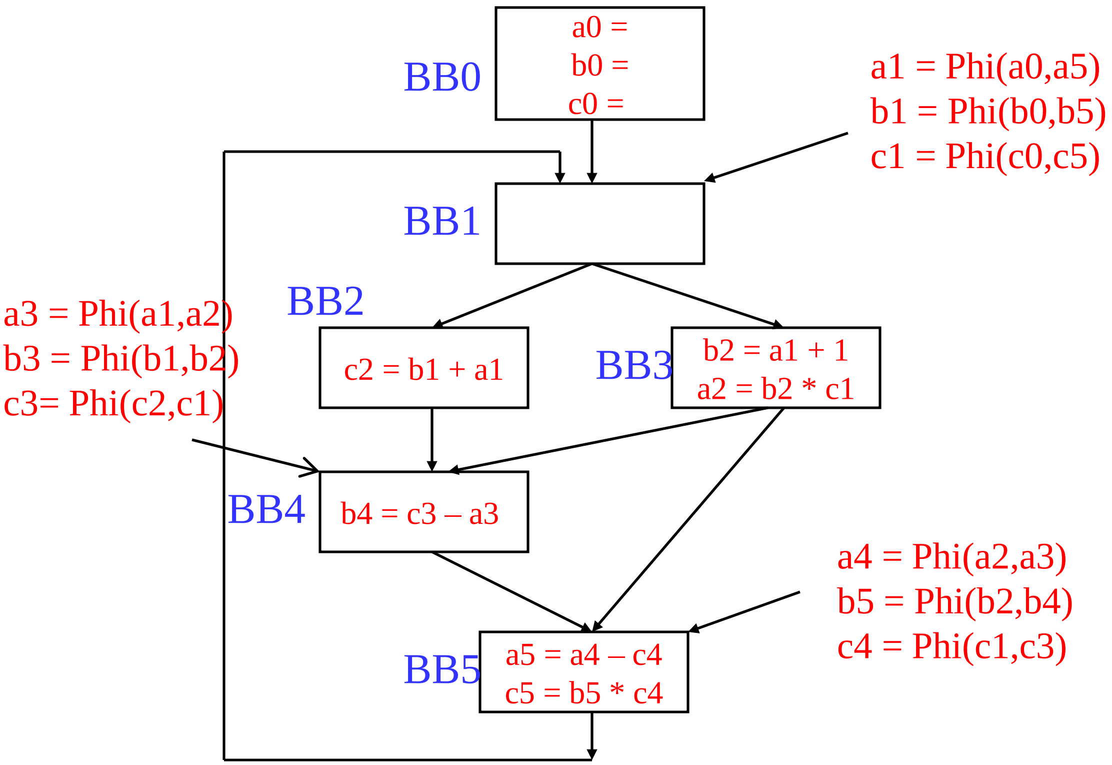
BB0

BB1

BB2 BB3 BB4 BB5

Homework Problem – Final Answer

Rename the variables



BB0

a0 =
b0 =
c0 =

a1 = Phi(a0,a5)
b1 = Phi(b0,b5)
c1 = Phi(c0,c5)

BB1

BB2

a3 = Phi(a1,a2)
b3 = Phi(b1,b2)
c3= Phi(c2,c1)

c2 = b1 + a1

BB3

b2 = a1 + 1
a2 = b2 * c1

BB4

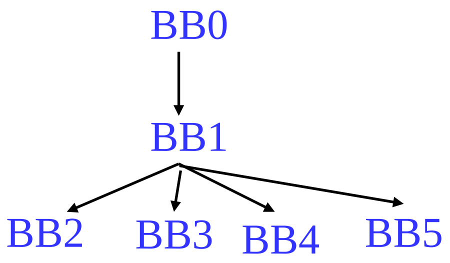
b4 = c3 – a3

a4 = Phi(a2,a3)
b5 = Phi(b2,b4)
c4 = Phi(c1,c3)

BB5

a5 = a4 – c4
c5 = b5 * c4

Dominator tree



Dominance frontier

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