EECS 583 – Class 5
Finish If-conversion, Start on Dataflow Analysis

University of Michigan

September 18, 2019

Reading Material + Announcements

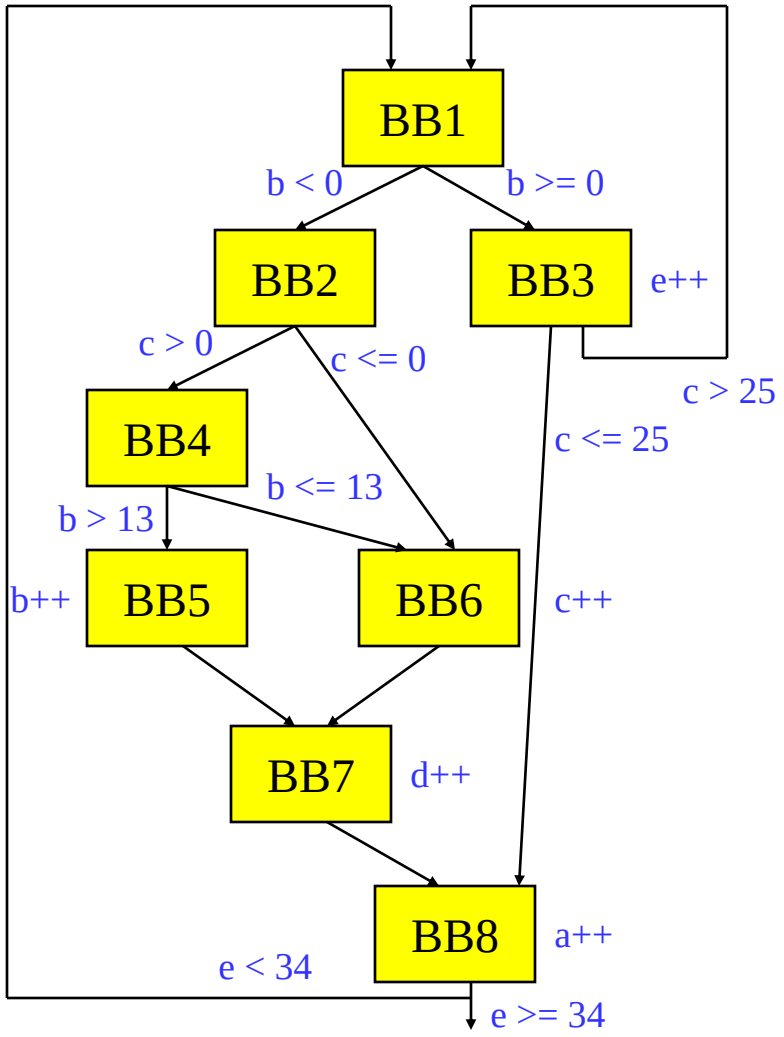
- Reminder – HW 1 due Friday at midnight
  - Submit uniquename_hw1.tgz file to:
    - eecs583a.eecs.umich.edu:/hw1_submissions
  - Before asking questions: 1) Read all threads on piazza, 2) Think a bit
    - Then, post question or talk to Sung if you are stuck
- Today's class
  - *Compilers: Principles, Techniques, and Tools*,
    A. Aho, R. Sethi, and J. Ullman, Addison-Wesley, 1988.
    (Chapters: 10.5, 10.6 Edition 1; Chapters 9.2 Edition 2)
- Material for next Monday
  - *Compilers: Principles, Techniques, and Tools*,
    A. Aho, R. Sethi, and J. Ullman, Addison-Wesley, 1988.
    (Chapters: 10.5, 10.6, 10.9, 10.10 Edition 1; Chapters 9.2, 9.3 Edition 2)

From Last Time: If-conversion

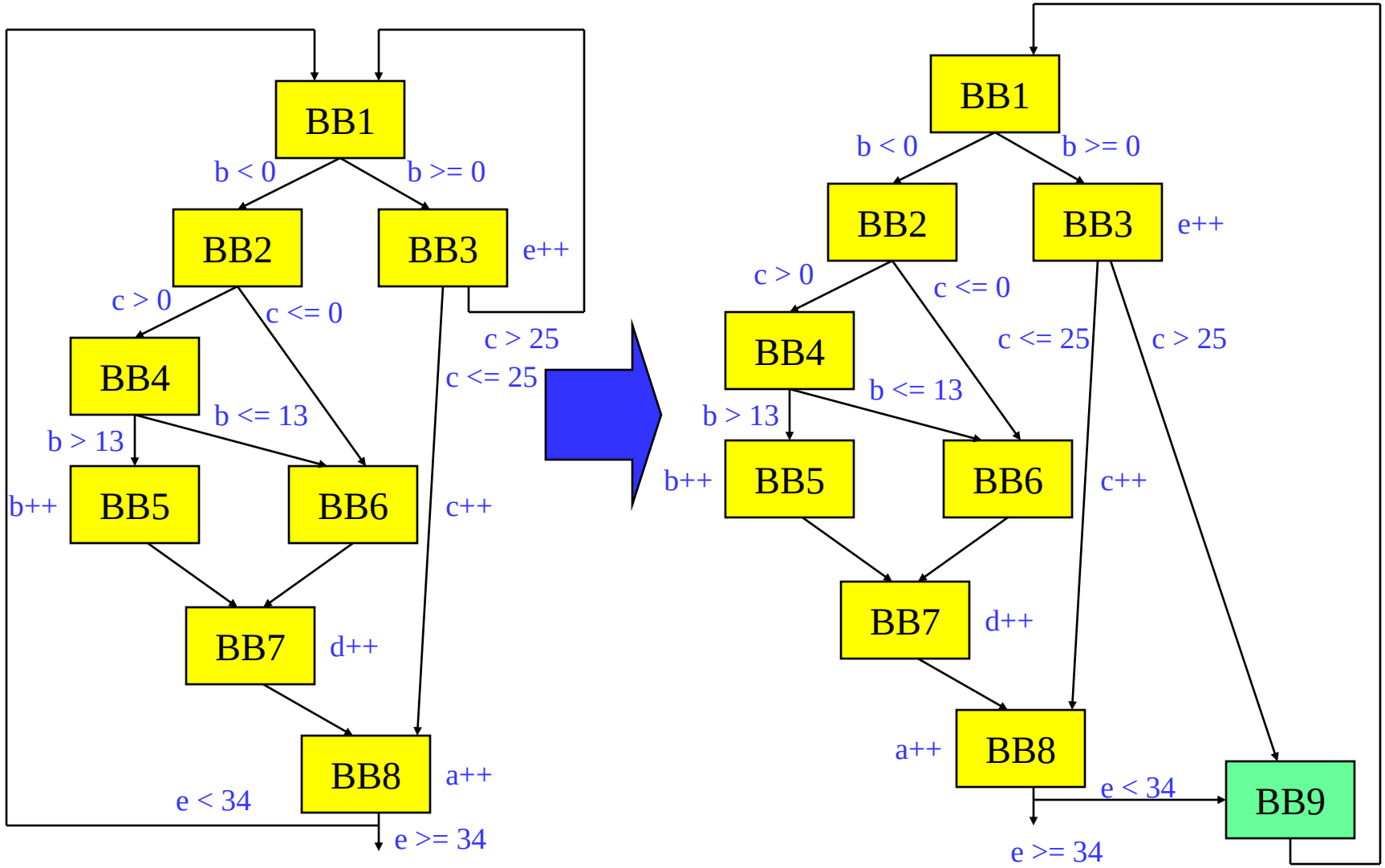
- Algorithm for generating predicated code
  - Automate what we've been doing by hand
  - Handle arbitrary complex graphs
    - But, acyclic subgraph only!!
    - Need a branch to get you back to the top of a loop
  - Efficient
- Roots are from Vector computer days
  - Vectorize a loop with an if-statement in the body
- 4 steps
  - 1. Loop backedge coalescing
  - 2. Control dependence analysis
  - 3. Control flow substitution
  - 4. CMPP compaction
- My version of Park & Schlansker

From Last Time: Running Example – Initial State

```
do {
    b = load(a)
    if (b < 0) {
        if ((c > 0) && (b > 13))
            b = b + 1
        else
            c = c + 1
        d = d + 1
    }
    else {
        e = e + 1
        if (c > 25) continue
    }
    a = a + 1
} while (e < 34)
```



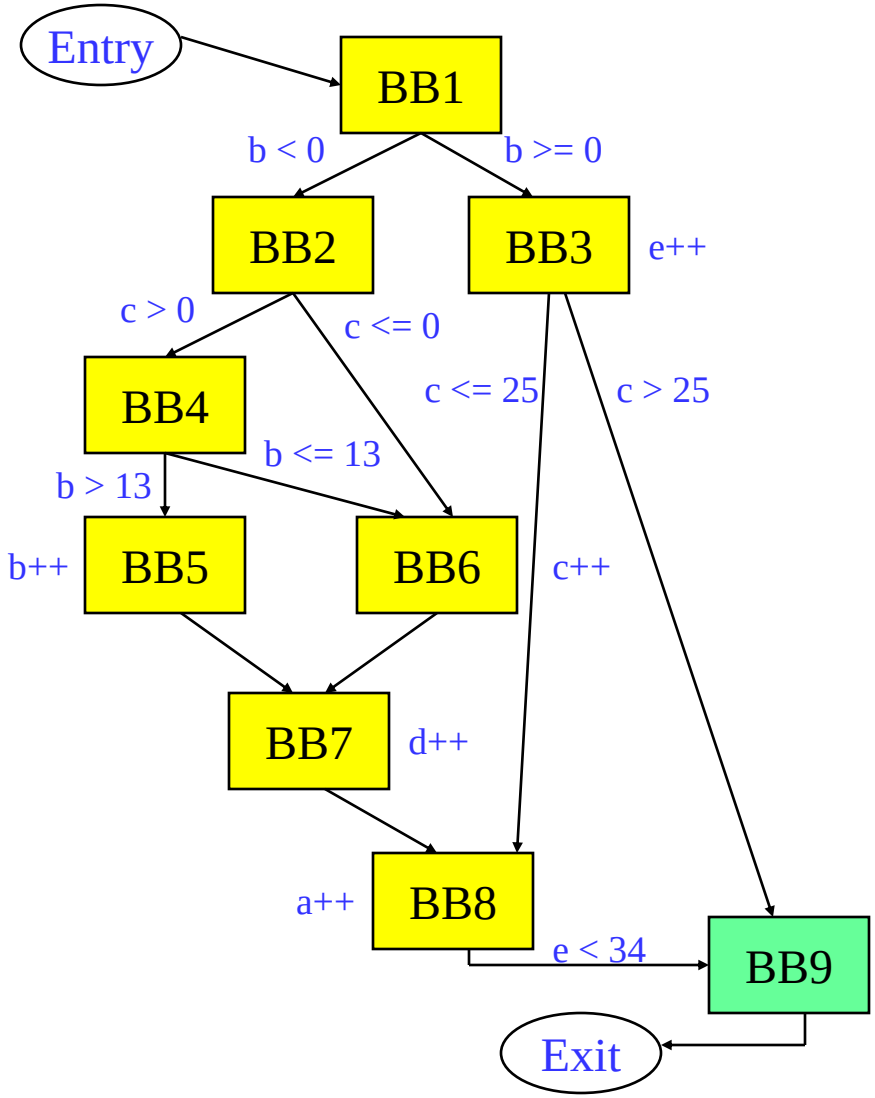
From Last Time: Running Example – Backedge Coalescing



From Last Time: Control Dependences

- Recall
  - Post dominator – BBX is post dominated by BBY if every path from BBX to EXIT contains BBY
  - Immediate post dominator – First breadth first successor of a block that is a post dominator
- Control dependence – BBY is control dependent on BBX iff
  - 1. There exists a directed path P from BBX to BBY with any BBZ in P (excluding BBX and BBY) post dominated by BBY
  - 2. BBX is not post dominated by BBY
- In English,
  - A BB is control dependent on the closest BB(s) that determine(s) its execution
  - Its actually not a BB, it's a control flow edge coming out of a BB

Running Example – CDs



First, nuke backedge(s)
Second, nuke exit edges
Then, Add pseudo entry/exit nodes
- Entry → nodes with no predecessors
- Exit → nodes with no successors

Control deps (left is taken)

BB1:
BB2:
BB3:
BB4:
BB5:
BB6:
BB7:
BB8:
BB9:

Algorithm for Control Dependence Analysis

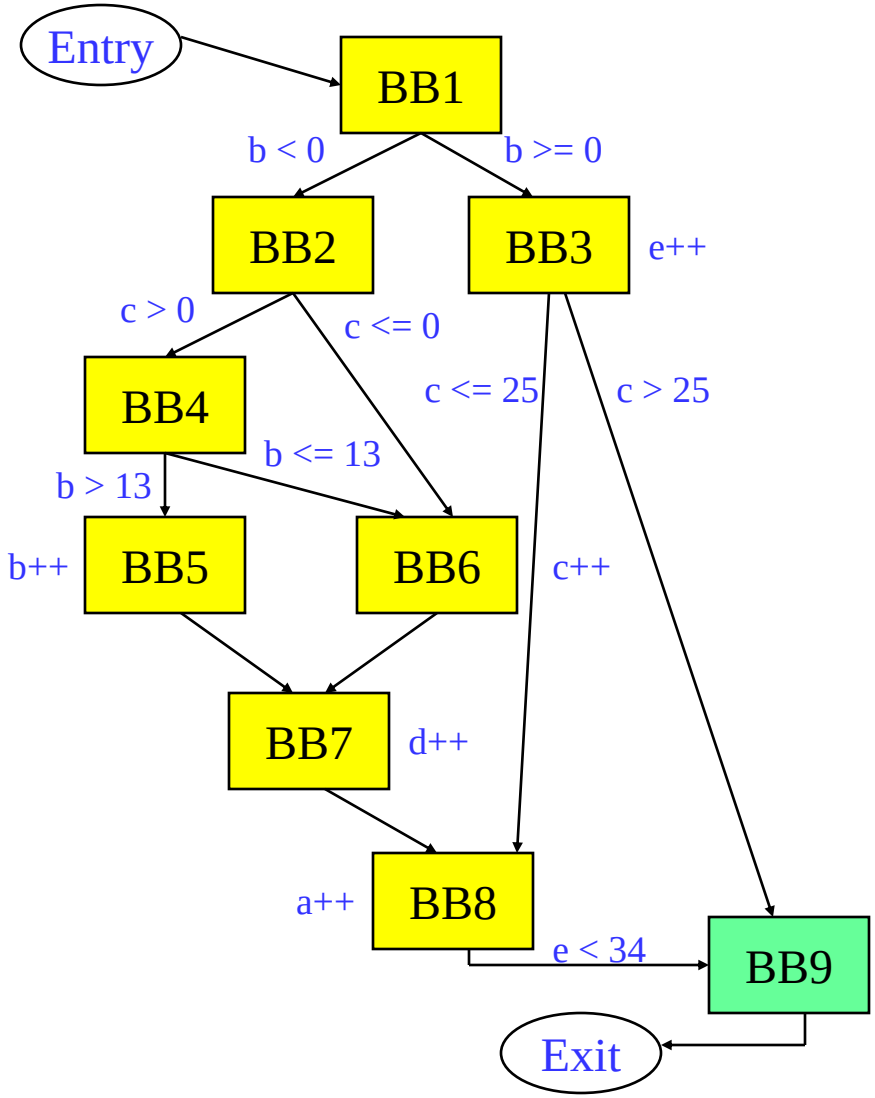
```
for each basic block x in region
   for each outgoing control flow edge e of x
      y = destination basic block of e
      if (y not in pdom(x)) then
         lub = ipdom(x)
         if (e corresponds to a taken branch) then
            x_id = -x.id
         else
            x_id = x.id
         endif
         t = y
         while (t != lub) do
            cd(t) += x_id;
            t = ipdom(t)
         endwhile
      endif
   endfor
endfor
```

Notes

Compute cd(x) which contains those BBs which x is control dependent on

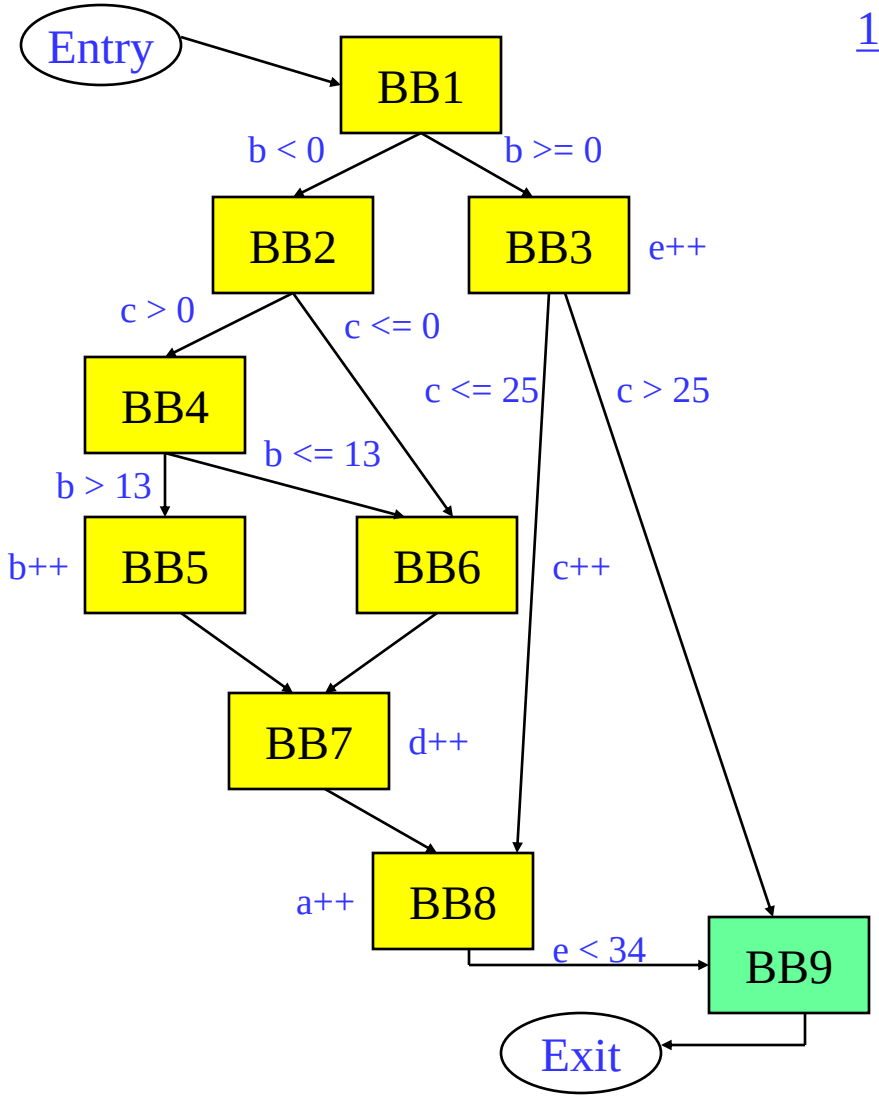
Iterate on per edge basis, adding edge to each cd set it is a member of

Running Example – Post Dominators



	pdom	ipdom
BB1:	1, 9, ex	9
BB2:	2, 7, 8, 9, ex	7
BB3:	3, 9, ex	9
BB4:	4, 7, 8, 9, ex	7
BB5:	5, 7, 8, 9, ex	7
BB6:	6, 7, 8, 9, ex	7
BB7:	7, 8, 9, ex	8
BB8:	8, 9, ex	9
BB9:	9, ex	ex

Running Example – CDs Via Algorithm

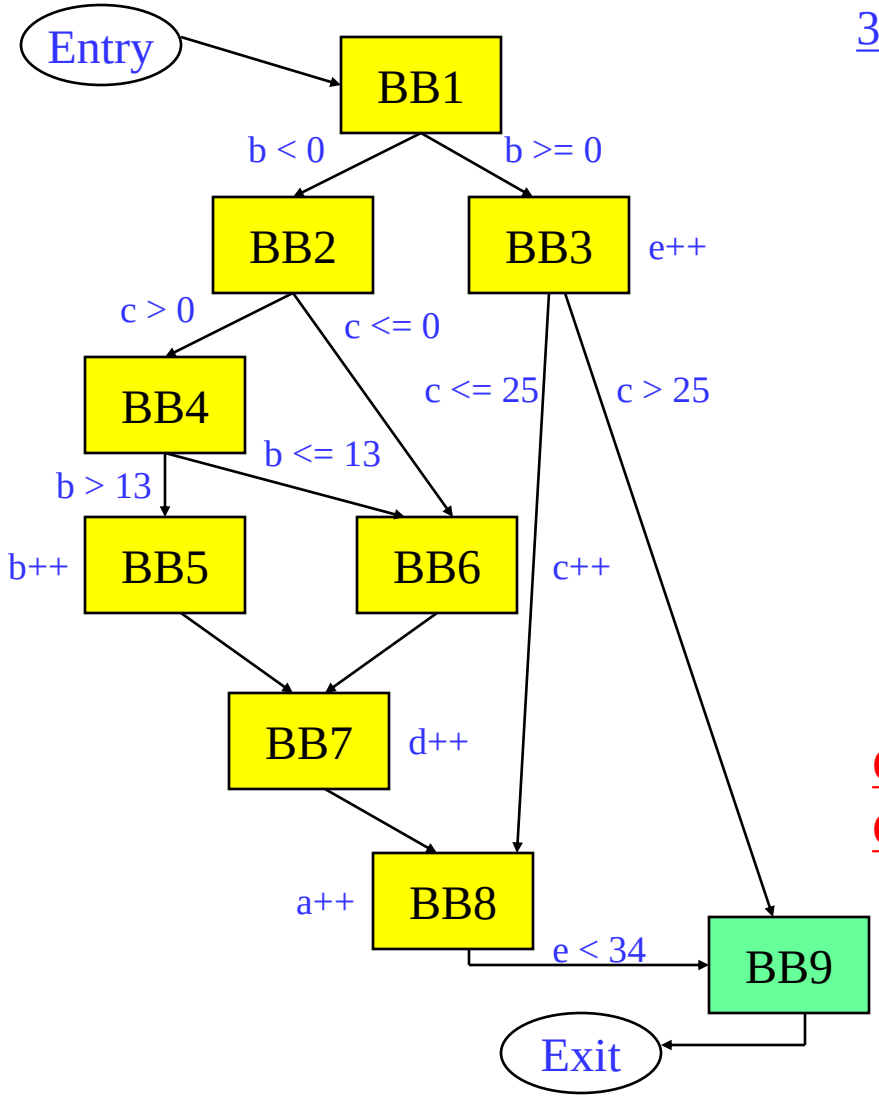


1 → 2 edge (aka –1)

x = 1
e = taken edge 1 → 2
y = 2
y not in pdom(x)
lub = 9
x_id = -1
t = 2

2 != 9 cd(2) += -1 t = 7
7 != 9 cd(7) += -1 t = 8
8 != 9 cd(8) += -1 t = 9
9 == 9

Running Example – CDs Via Algorithm (2)



3 → 8 edge (aka -3)

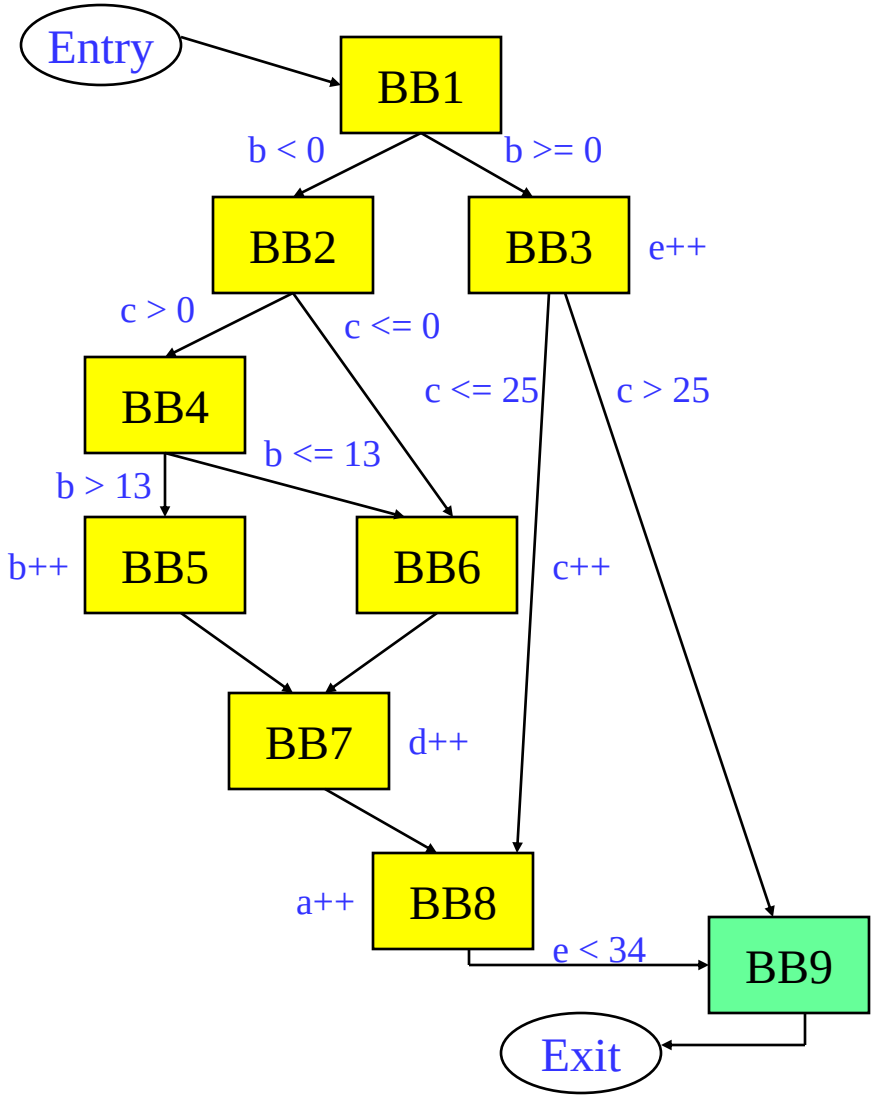
x = 3
e = taken edge 3 → 8
y = 8
y not in pdom(x)
lub = 9
x_id = -3
t = 8

8 != 9 cd(8) += -3 t = 9
9 == 9

Class ProblemA: 1 → 3 edge (aka 1)

Class ProblemB: 7 → 8 edge (aka -7)

Running Example – CDs Via Algorithm (3)



Control deps (left is taken)

BB1: none

BB2: -1

BB3: 1

BB4: -2

BB5: -4

BB6: 2, 4

BB7: -1

BB8: -1, -3

BB9: none

Step 3: Control Flow Substitution

- Go from branching code → sequential predicated code
- 5 baby steps
  - 1. Create predicates
  - 2. CMPP insertion
  - 3. Guard operations
  - 4. Remove branches
  - 5. Initialize predicates

Predicate Creation

- R/K calculation – Mapping predicates to blocks
  - Paper more complicated than it really is
  - K = unique sets of control dependences
  - Create a new predicate for each element of K
  - R(bb) = predicate that represents CD set for bb, ie the bb's assigned predicate (all ops in that bb guarded by R(bb))

K = {{-1}, {1}, {-2}, {-4}, {2,4}, {-1,-3}}
predicates = p1, p2, p3, p4, p5, p6

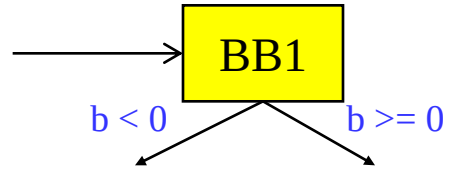
bb	=	1	2	3	4	5	6	7	8	9
CD(bb)	=	{none}	{-1}	{1}	{-2}	{-4}	{2,4}	{-1}	{-1,-3}	{none}
R(bb)	=	T	p1	p2	p3	p4	p5	p1	p6	T

CMPP Creation/Insertion

- For each control dependence set
  - For each edge in the control dependence set
    - Identify branch condition that causes edge to be traversed
    - Create CMPP to compute corresponding branch condition
      - OR-type – handles worst case
      - guard = True
      - destination = predicate assigned to that CD set
      - Insert at end of BB that is the source of the edge

K = {{-1}, {1}, {-2}, {-4}, {2,4}, {-1,-3}}
predicates = p1, p2, p3, p4, p5, p6

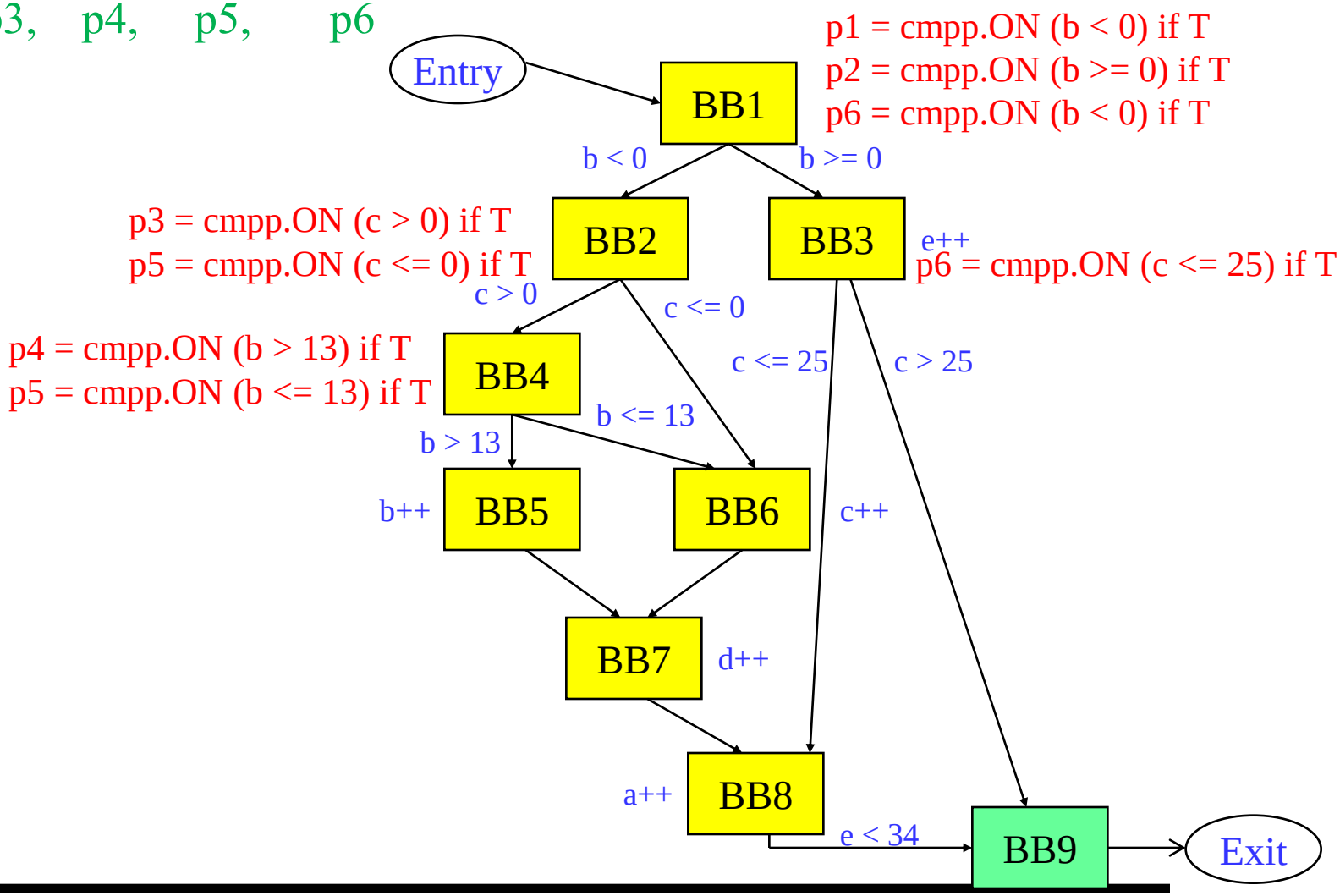
Example: p1 = cmpp.ON (b < 0) if T



Running Example – CMPP Creation

K = {{-1}, {1}, {-2}, {-4}, {2,4}, {-1,-3}}

p's = p1, p2, p3, p4, p5, p6

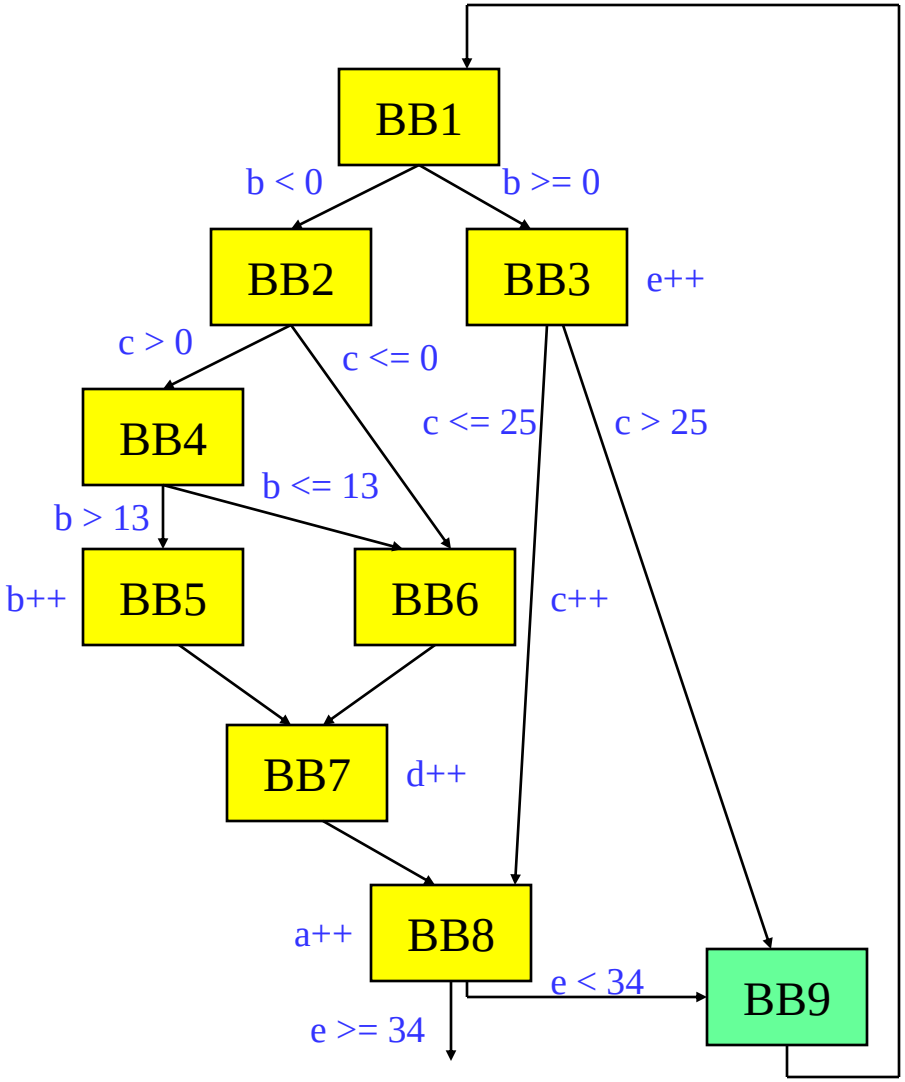


Control Flow Substitution – The Rest

- Guard all operations in each bb by R(bb)
  - » Including the newly inserted CMPPs
- Nuke all the branches
  - » Except exit edges and backedges
- Initialize each predicate to 0 in first BB

bb	=	1,	2,	3,	4,	5,	6,	7,	8,	9
CD(bb)	=	{{none},	{-1},	{1},	{-2},	{-4},	{2,4},	{-1},	{-1,-3},	{none}
R(bb)	=	T	p1	p2	p3	p4	p5	p1	p6	T

Running Example – Control Flow Substitution



Loop:

p1 = p2 = p3 = p4 = p5 = p6 = 0
b = load(a) if T
p1 = cmpp.ON (b < 0) if T
p2 = cmpp.ON (b >= 0) if T
p6 = cmpp.ON (b < 0) if T
p3 = cmpp.ON (c > 0) if p1
p5 = cmpp.ON (c <= 0) if p1
p4 = cmpp.ON (b > 13) if p3
p5 = cmpp.ON (b <= 13) if p3
b = b + 1 if p4
c = c + 1 if p5
d = d + 1 if p1
p6 = cmpp.ON (c <= 25) if p2
e = e + 1 if p2
a = a + 1 if p6
bge e, 34, Done if p6
jump Loop if T

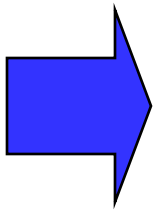
Done:

Step 4: CMPP Compaction

- Convert ON CMPPs to UN
  - All singly defined predicates don't need to be OR-type
  - OR of 1 condition → Just compute it !!!
  - Remove initialization (Unconditional don't require init)
- Reduce number of CMPPs
  - Utilize 2nd destination slot
  - Combine any 2 CMPPs with:
    - Same source operands
    - Same guarding predicate
    - Same or opposite compare conditions

Running Example - CMPP Compaction

```
Loop:
   p1 = p2 = p3 = p4 = p5 = p6 = 0
   b = load(a) if T
   p1 = cmpp.ON (b < 0) if T
   p2 = cmpp.ON (b >= 0) if T
   p6 = cmpp.ON (b < 0) if T
   p3 = cmpp.ON (c > 0) if p1
   p5 = cmpp.ON (c <= 0) if p1
   p4 = cmpp.ON (b > 13) if p3
   p5 = cmpp.ON (b <= 13) if p3
   b = b + 1 if p4
   c = c + 1 if p5
   d = d + 1 if p1
   p6 = cmpp.ON (c <= 25) if p2
   e = e + 1 if p2
   a = a + 1 if p6
   bge e, 34, Done if p6
   jump Loop if T
Done:
```



```
Loop:
   p5 = p6 = 0
   b = load(a) if T
   p1,p2 = cmpp.UN.UC (b < 0) if T
   p6 = cmpp.ON (b < 0) if T
   p3,p5 = cmpp.UN.OC (c > 0) if p1
   p4,p5 = cmpp.UN.OC (b > 13) if p3
   b = b + 1 if p4
   c = c + 1 if p5
   d = d + 1 if p1
   p6 = cmpp.ON (c <= 25) if p2
   e = e + 1 if p2
   a = a + 1 if p6
   bge e, 34, Done if p6
   jump Loop if T
Done:
```

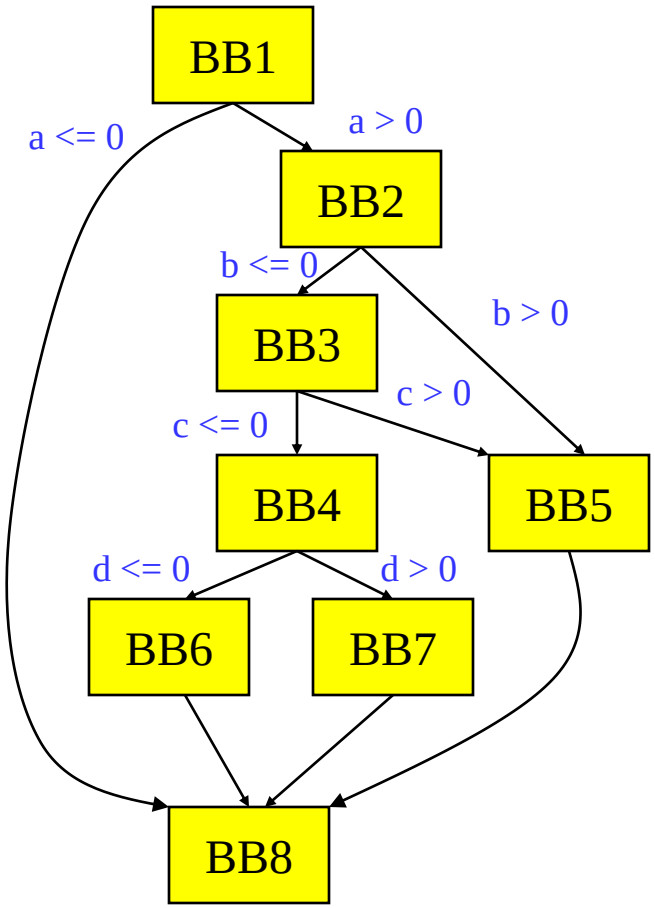
Homework Problem – Answer on Next Slide

```
if (a > 0) {
    r = t + s
    if (b > 0 || c > 0)
        u = v + 1
    else if (d > 0)
        x = y + 1
    else
        z = z + 1
}
```

a. Draw the CFG
b. Compute CD
c. If-convert the code

Homework Problem Answer

```
if (a > 0) {
   r = t + s
   if (b > 0 || c > 0)
      u = v + 1
   else if (d > 0)
      x = y + 1
   else
      z = z + 1
}
```



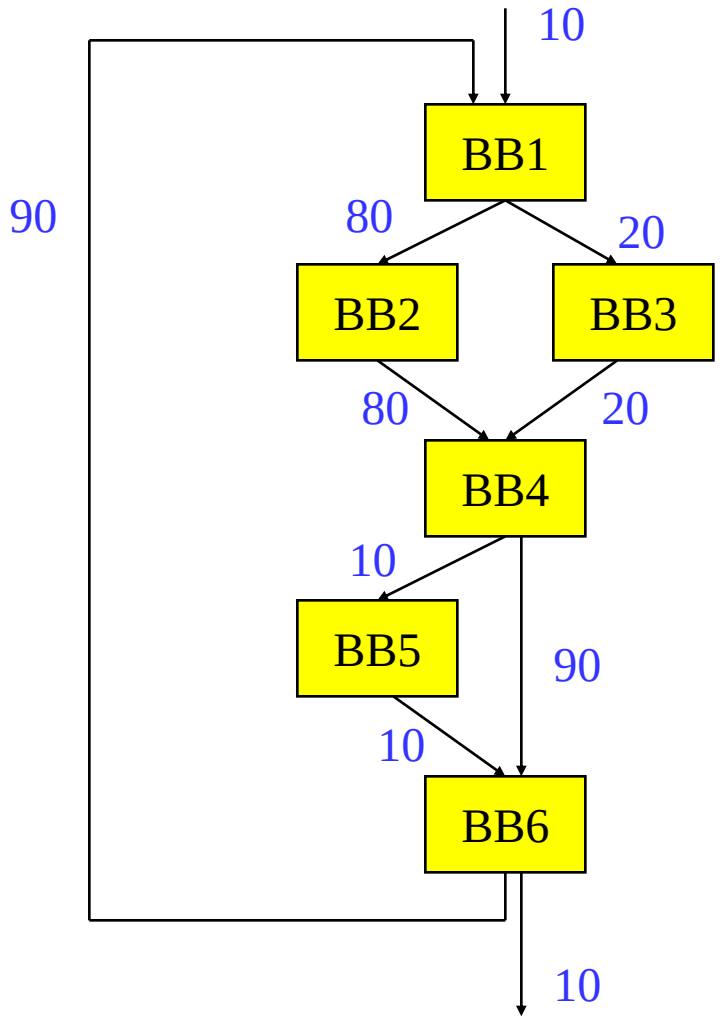
BB	CD
1	-
2	1
3	-2
4	-3
5	2,3
6	-4
7	4
8	-

a. Draw the CFG
b. Compute CD
c. If-convert the code

```
p3 = 0
p1 = CMPP.UN (a > 0) if T
r = t + s if p1
p2,p3 = CMPP.UC.ON (b > 0) if p1
p4,p3 = CMPP.UC.ON (c > 0) if p2
u = v + 1 if p3
p5,p6 = CMPP.UC.UN (d > 0) if p4
x = y + 1 if p6
z = z + 1 if p5
```

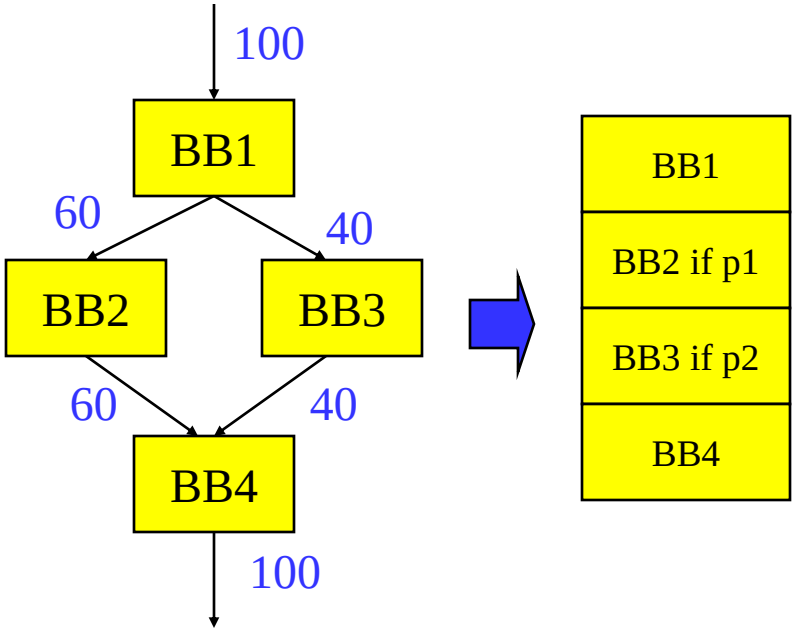
When to Apply If-conversion?

- Positives
  - Remove branch
    - No disruption to sequential fetch
    - No prediction or mispredict
    - No draining of pipeline for mispredict
    - No use of branch resource
  - Increase potential for operation overlap
    - Creates larger basic blocks
    - Convert control dependences into data dependences
  - Enable more aggressive compiler xforms
    - Software pipelining
    - Height reduction



Negative 1: Resource Usage

Instruction execution is additive for all BBs that are if-converted, thus require more processor resources



Case 1: Each BB requires 3 resources
Assume processor has **2** resources

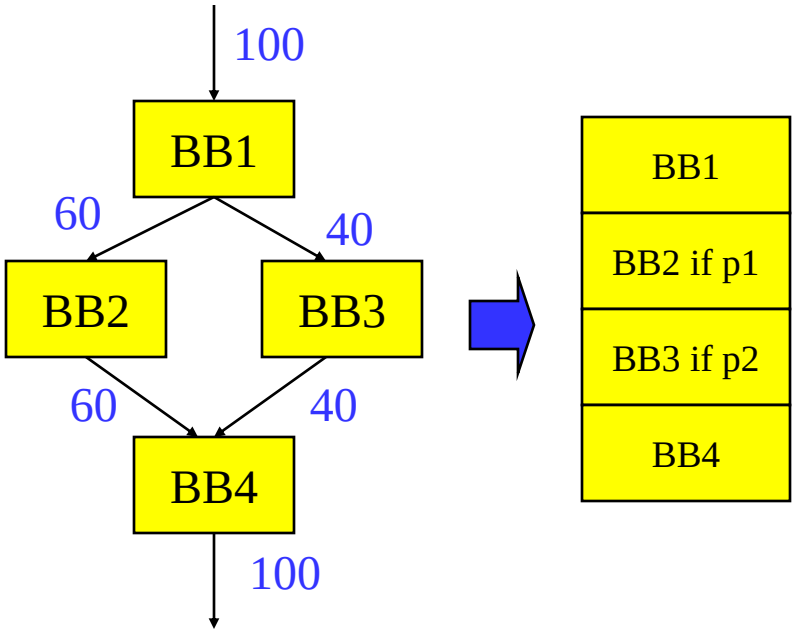
No IC: 1*3 + .6*3 + .4*3 + 1*3 = 9
9 / 2 = 4.5 = 5 cycles
IC: 1(3 + 3 + 3+ 3) = 12
12 / 2 = 6 cycles

Case 2: Each BB requires 3 resources
Assume processor has **6** resources

No IC: 1*3 + .6*3 + .4*3 + 1*3 = 9
9 / 6 = 1.5 = 2 cycles
IC: 1(3+3+3+3) = 12
12 / 6 = 2 cycles

Negative 2: Dependence Height

Dependence height is max of
for all BBs that are if-converted
(dep height = schedule length
with infinite resources)



Case 1: height(bb1) = 1, height(bb2) = 3
Height(bb3) = **9**, height(bb4) = 2

No IC: 1*1 + .6*3 + .4*9 + 1*2 = 8.4

IC: 1*1 + 1*MAX(3,9) + 1*3 = 13

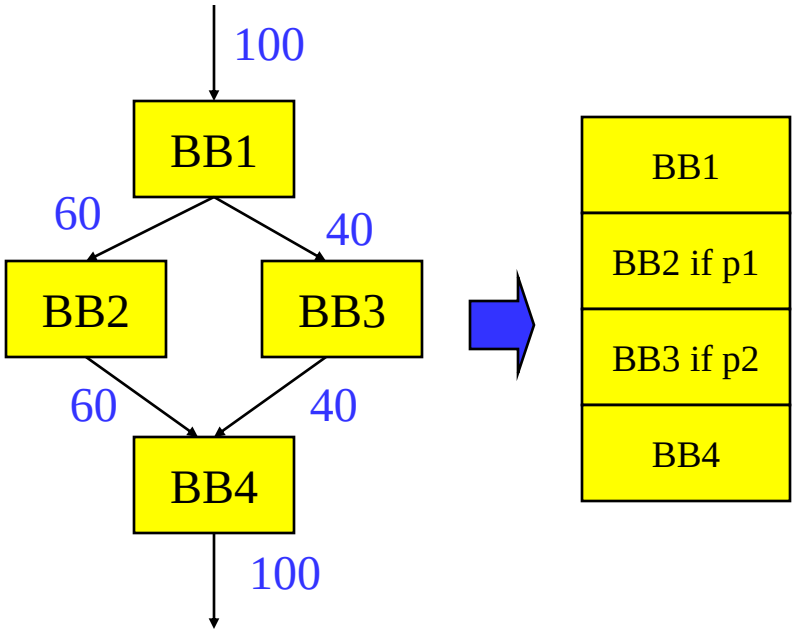
Case 2: height(bb1) = 1, height(bb2) = 3
Height(bb3) = **3**, height(bb4) = 2

No IC: 1*1 + .6*3 + .4*3 + 1*2 = 6

IC: 1*1 + 1*MAX(3,3) + 1*2 = 6

Negative 3: Hazard Presence

Hazard = operation that forces
the compiler to be conservative,
so limited reordering or optimization,
e.g., subroutine call, pointer store, …



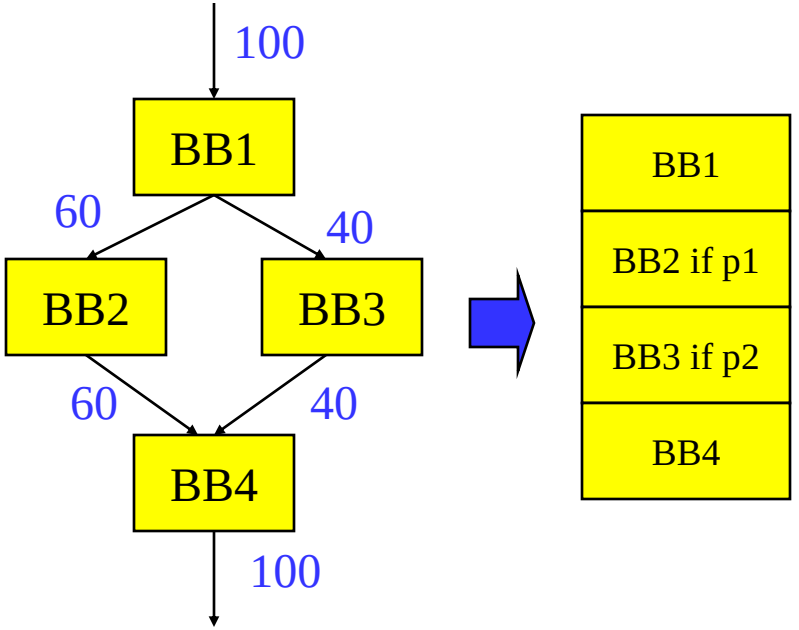
Case 1: Hazard in BB3

No IC : SB out of BB1, 2, 4, operations
In BB4 free to overlap with those in
BB1 and BB2

IC: operations in BB4 cannot overlap
With those in BB1 (BB2 ok)

Deciding When/What To If-convert

- Resources
  - Small resource usage ideal for less important paths
- Dependence height
  - Matched heights are ideal
  - Close to same heights is ok
- Remember everything is relative for resources and dependence height !
- Hazards
  - Avoid hazards unless on most important path
- Estimate of benefit
  - Branches/Mispredicts removed
  - Fudge factor
- Read more about Hyperblock Formation if you are interested

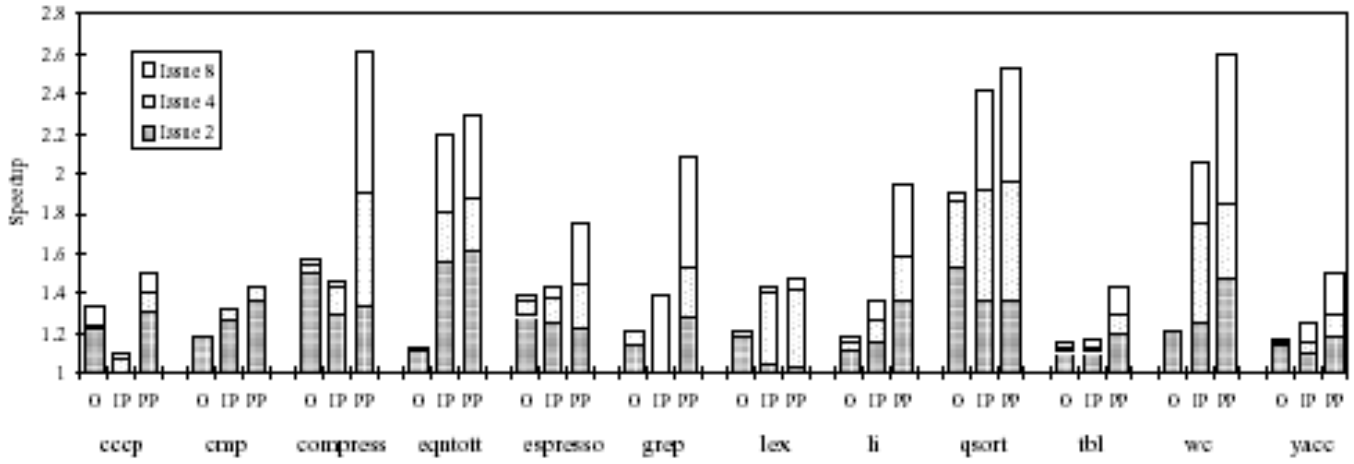


For More on If-conversion

- See
  - "Effective Compiler Support for Predicated Execution using the Hyperblock", S. Mahlke et al., MICRO-25, 1992.
  - "Control CPR: A Branch Height Reduction Optimization for EPIC Processors", M. Schlansker et al., PLDI-99, 1999.

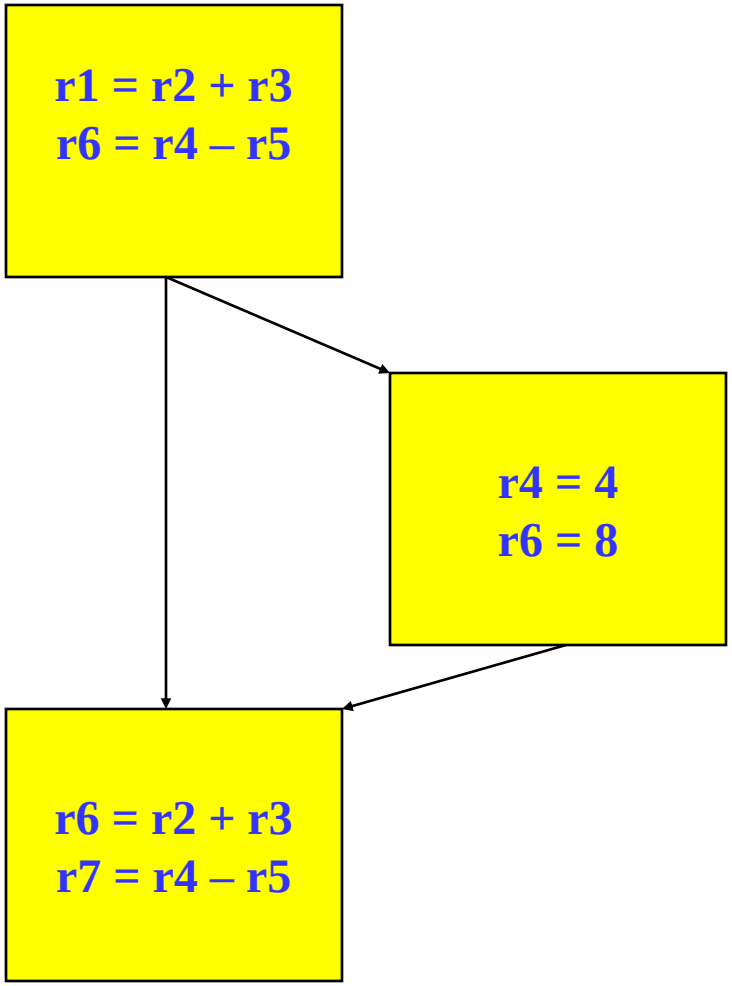
Hyperblock Performance Evaluation

- O = BB code
- IP = Structural if-conversion
  - All innermost loops, acyclic SEME regions
- PP = Selective if-conversion



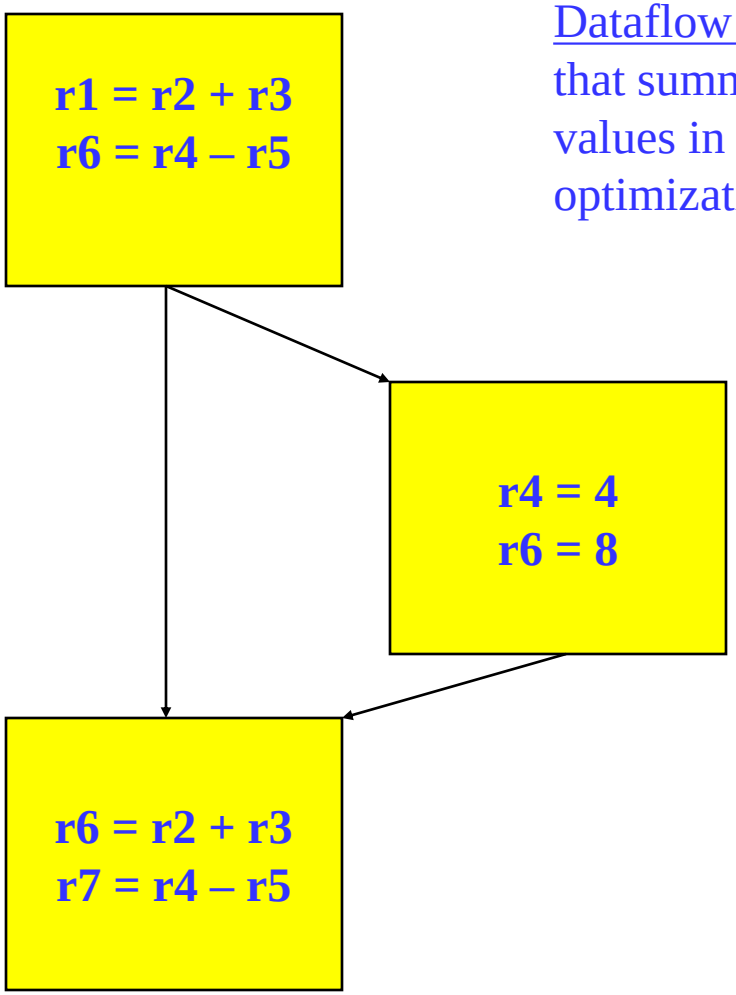
New Topic
Dataflow Analysis!

Looking Inside the Basic Blocks: Dataflow Analysis + Optimization



- Control flow analysis
  - Treat BB as black box
  - Just care about branches
- Now
  - Start looking at ops in BBs
  - What's computed and where
- Classical optimizations
  - Want to make the computation more efficient
- Ex: Common Subexpression Elimination (CSE)
  - Is r2 + r3 redundant?
  - Is r4 – r5 redundant?
  - What if there were 1000 BB's
  - Dataflow analysis !!

Dataflow Analysis Introduction



Dataflow analysis – Collection of information that summarizes the creation/destruction of values in a program. Used to identify legal optimization opportunities.

Pick an arbitrary point in the program

Which VRs contain useful data values? (liveness or upward exposed uses)

Which definitions may reach this point? (reaching defns)

Which definitions are guaranteed to reach this point? (available defns)

Which uses below are exposed? (downward exposed uses)

Live Variable (Liveness) Analysis

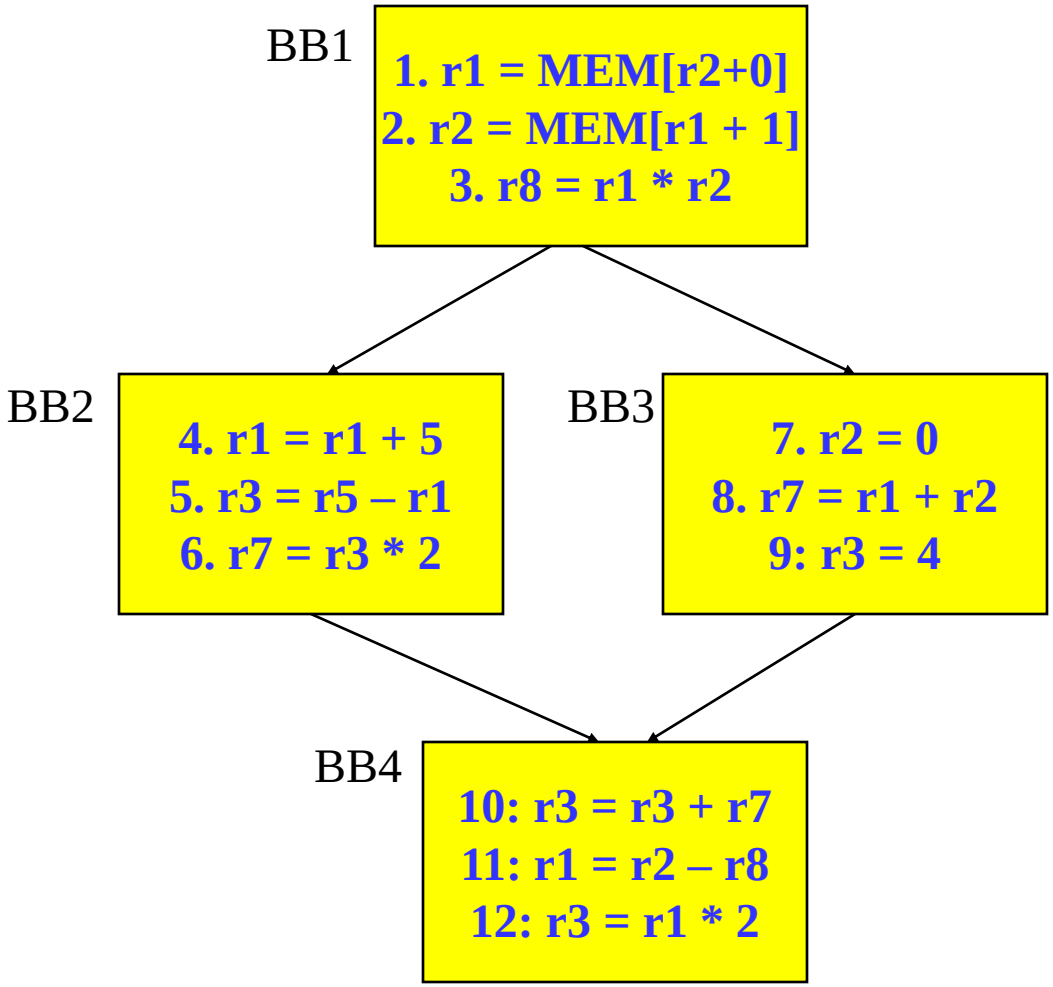
- Defn: For each point p in a program and each variable y, determine whether y can be used before being redefined starting at p
- Algorithm sketch
  - For each BB, y is live if it is used before defined in the BB or it is live leaving the block
  - Backward dataflow analysis as propagation occurs from uses upwards to defs
- 4 sets
  - GEN = set of external variables consumed in the BB
  - KILL = set of external variable uses killed by the BB
    - equivalent to set of variables defined by the BB
  - IN = set of variables that are live at the entry point of a BB
  - OUT = set of variables that are live at the exit point of a BB

Computing GEN/KILL Sets For Each BB

```
for each basic block in the procedure, X, do
    GEN(X) = 0
    KILL(X) = 0
    for each operation in reverse sequential order in X, op, do
        for each destination operand of op, dest, do
            GEN(X) -= dest
            KILL(X)  += dest
        endfor
        for each source operand of op, src, do
            GEN(X) += src
            KILL(X) -= src
        endfor
    endfor
endfor
```

Example – GEN/KILL Liveness Computation

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



Compute IN/OUT Sets for all BBs

```
initialize IN(X) to 0 for all basic blocks X
change = 1
while (change) do
    change = 0
    for each basic block in procedure, X, do
        old_IN = IN(X)
        OUT(X) = Union(IN(Y)) for all successors Y of X
        IN(X) = GEN(X) + (OUT(X) – KILL(X))
        if (old_IN != IN(X)) then
            change = 1
        endif
    endfor
endfor
```

Example – Liveness Computation

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