# EECS 583 – Class 2
# Control Flow Analysis

*University of Michigan*

*January 17, 2024*

https://web.eecs.umich.edu/~mahlke/courses/583w24

# Announcements & Reading Material

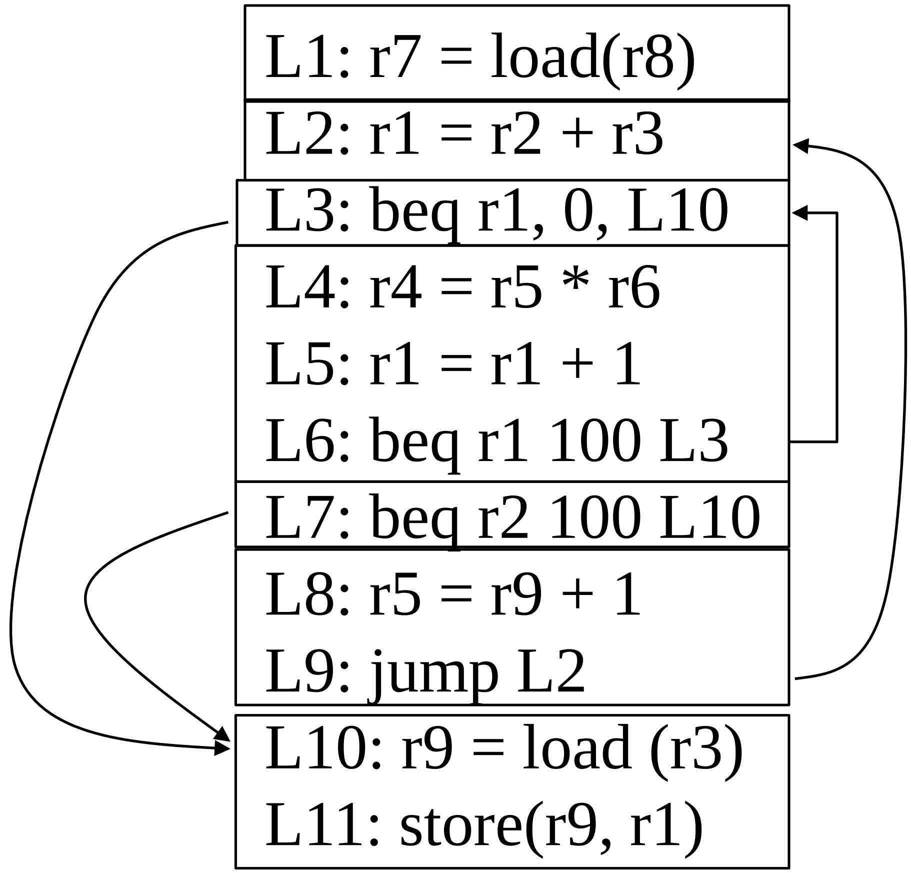
- eecs583a,eecs583b.eecs.umich.edu
  - Everyone has home directory and login
  - Let us know if you can't login (VPN required!)
- HW 0 – Due today, but nothing to turn in (needed for HW1)
  - Please get this done ASAP, talk to Aditya/Yunjie if you have problems
- Reading
  - Today's class
    - Ch 9.4, 10.4 (6.6, 9.6) from Compilers: Principles, Techniques Tools Ed 1 (Ed 2)
  - Next class
    - "Trace Selection for Compiling Large C Applications to Microcode", Chang and Hwu, MICRO-21, 1988.
    - "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", Hwu et al., Journal of Supercomputing, 1993

# Homework 1 – Due Mon Jan 29

- Get started ASAP. If you haven't done HW0, you are behind!
- Goals: Learn how to profile with LLVM, write stats collection pass
- 583_W24_HW1.tgz
  - hw1pass.cpp: template for your pass
  - 583simple, 583anagram, 583compress: benchmark source code + inputs + expected outputs + run instructions
- Easy to do, but hard to start because of newness
  - Look for Aditya's piazza post for help, more info in today's discussion section
    - Skeleton code
    - How to run profiler
    - Simple example with opcode stats
  - Talk to the GSIs if you are stuck

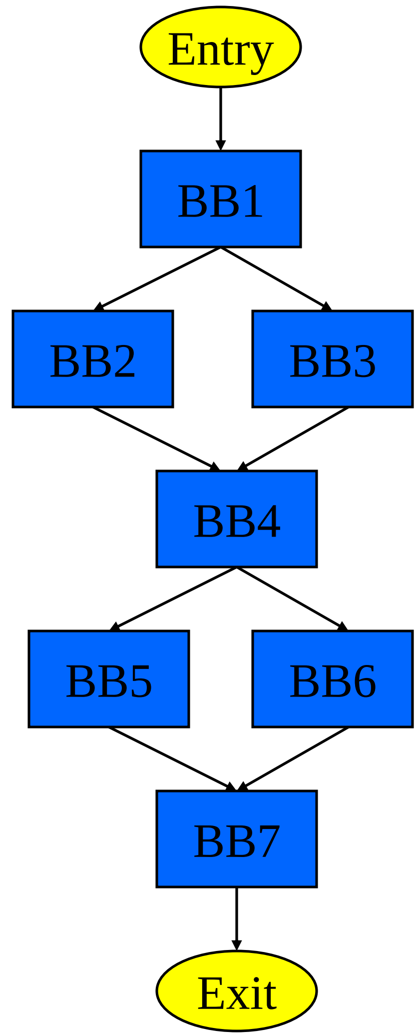
# From Last Time: Identifying BBs - Answer

```
L1: r7 = load(r8)
L2: r1 = r2 + r3
L3: beq r1, 0, L10
L4: r4 = r5 * r6
L5: r1 = r1 + 1
L6: beq r1 100 L3
L7: beq r2 100 L10
L8: r5 = r9 + 1
L9: jump L2
L10: r9 = load (r3)
L11: store(r9, r1)
```



# From Last Time: Control Flow Graph (CFG)

- Defn Control Flow Graph – Directed graph, G = (V,E) where each vertex V is a basic block and there is an edge E, v1 (BB1) → v2 (BB2) if BB2 can immediately follow BB1 in some execution sequence
  - » A BB has an edge to all blocks it can branch to
  - » Standard representation used by many compilers
  - » Often have 2 pseudo vertices
    - entry node
    - exit node

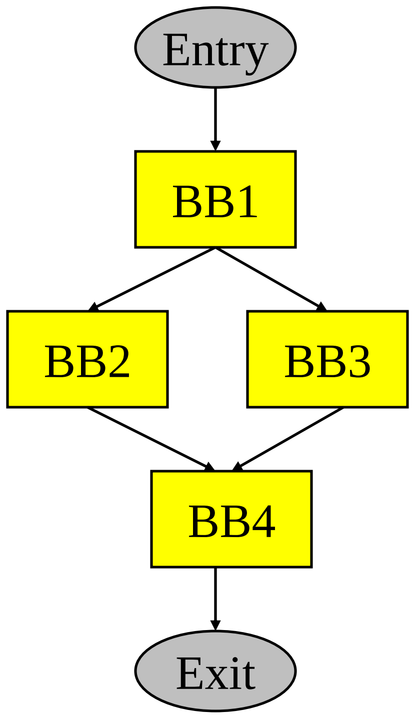


# From Last Time: Property of CFGs: Dominator (DOM)

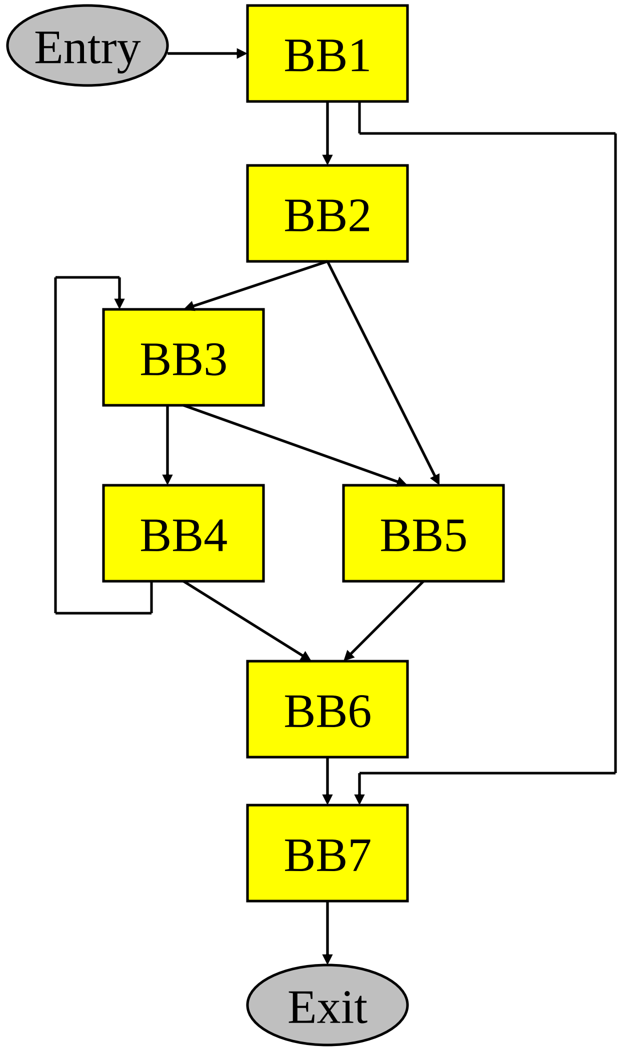
- Defn: Dominator – Given a CFG(V, E, Entry, Exit), a node x dominates a node y, if every path from the Entry block to y contains x
- 3 properties of dominators
  - Each BB dominates itself
  - If x dominates y, and y dominates z, then x dominates z
  - If x dominates z and y dominates z, then either x dominates y or y dominates x
- Intuition
  - Given some BB, which blocks are guaranteed to have executed prior to executing the BB

# Dominator Example 1

Compute Dom(BBi) = set of blocks that dominate BBi



# Dominator Example 2

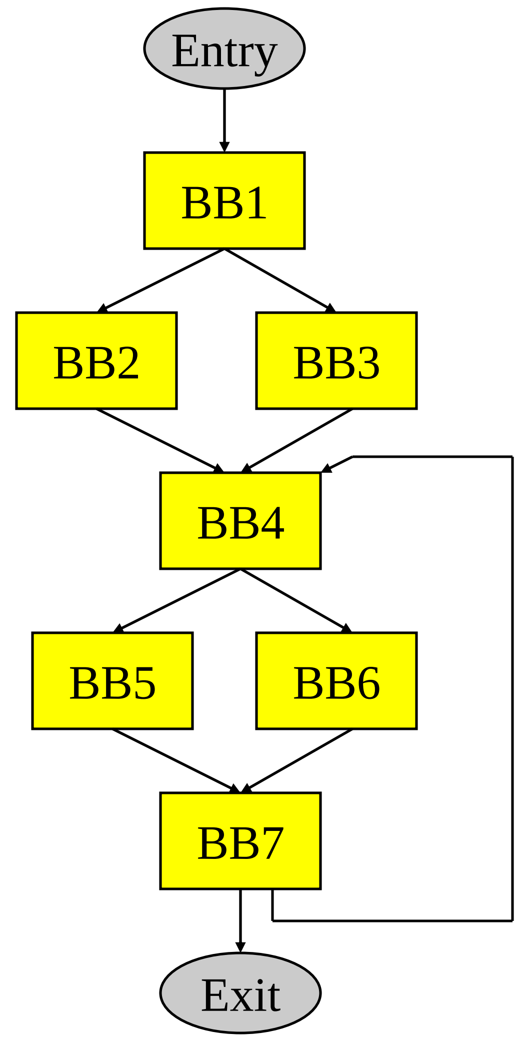


# Dominator Analysis

- Compute dom(BBi) = set of BBs that dominate BBi
- Initialization
  - » Dom(entry) = entry
  - » Dom(everything else) = all nodes
- Iterative computation
  - » while change, do
    - change = false
    - for each BB (except the entry BB)
      - tmp(BB) = BB + {intersect of Dom of all predecessor BB's}
      - if (tmp(BB) != dom(BB))

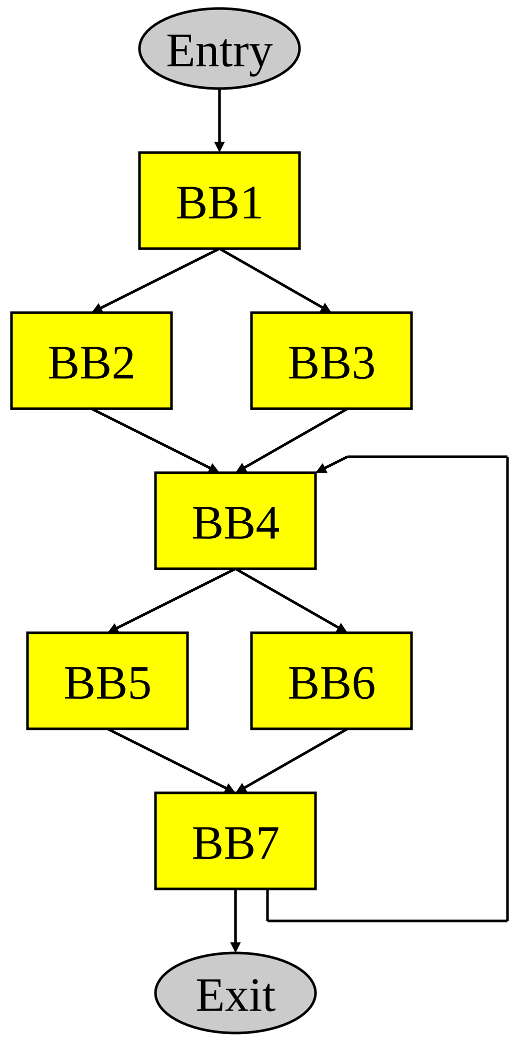
        dom(BB) = tmp(BB)

        change = true



# Immediate Dominator

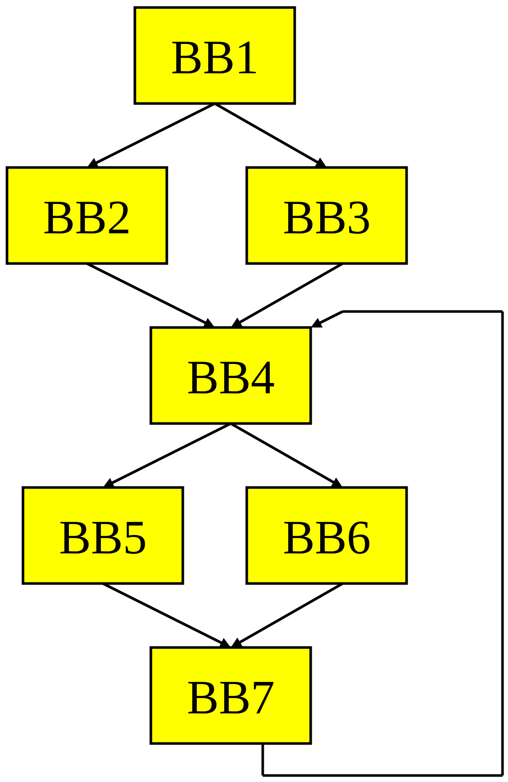
- Defn: Immediate dominator (idom) – Each node n has a unique immediate dominator m that is the last dominator of n on any path from the initial node to n
  - Closest node that dominates

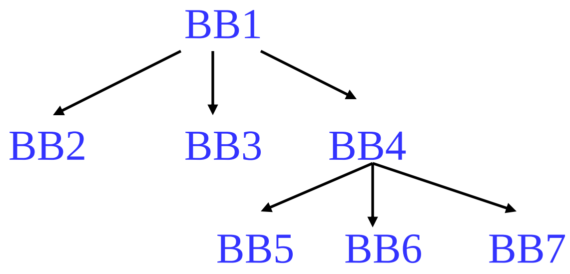


# Dominator Tree

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

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



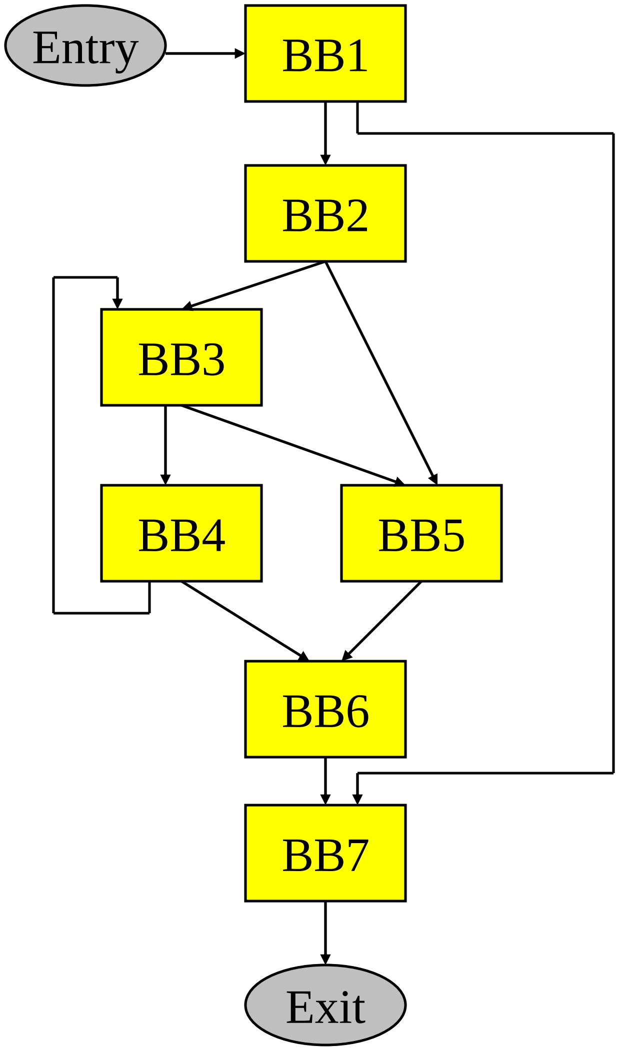




**Dom tree**

# Dominator Tree Example

Draw the dominator tree

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

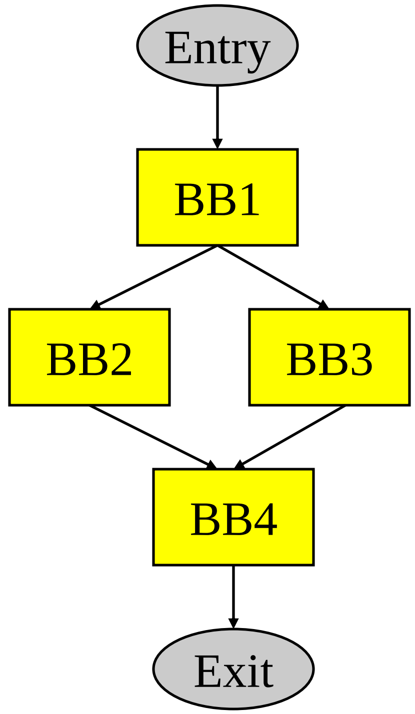


# Post Dominator (PDOM)

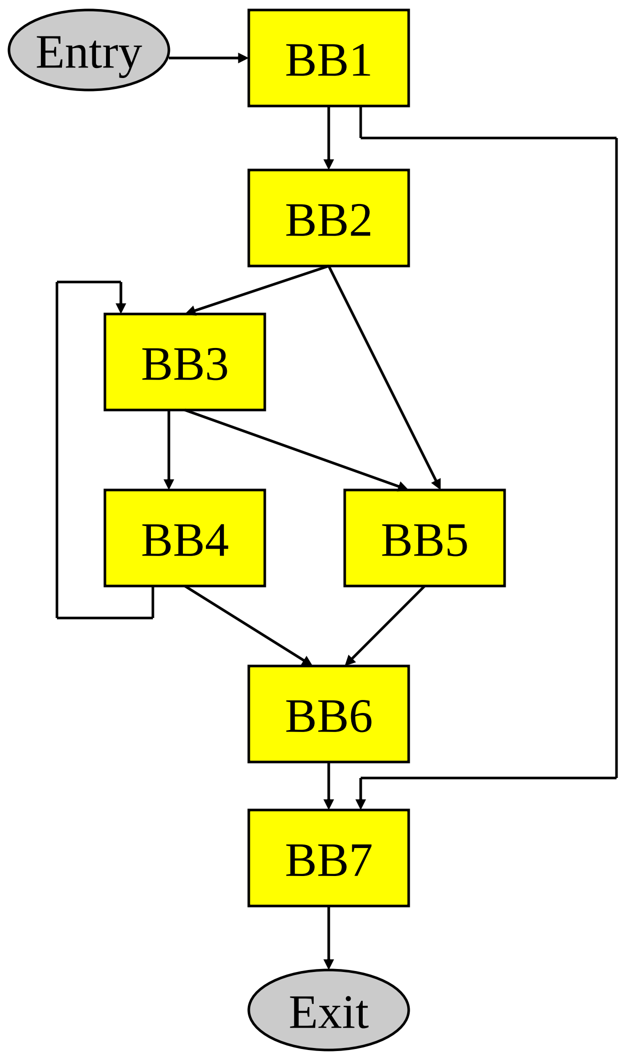
- Reverse of dominator
- Defn: Post Dominator – Given a CFG(V, E, Entry, Exit), a node x post dominates a node y, if every path from y to the Exit contains x
- Intuition
  - Given some BB, which blocks are guaranteed to have executed after executing the BB
- pdom(BBi) = set of BBs that post dominate BBi

- Initialization
  - Pdom(exit) = exit
  - Pdom(everything else) = all nodes
- Iterative computation
  - while change, do
    - change = false
    - for each BB (except the exit BB)
      - tmp(BB) = BB + {intersect of pdom of all successor BB's}
      - if (tmp(BB) != pdom(BB))
        pdom(BB) = tmp(BB)
        change = true

# Post Dominator Example 1

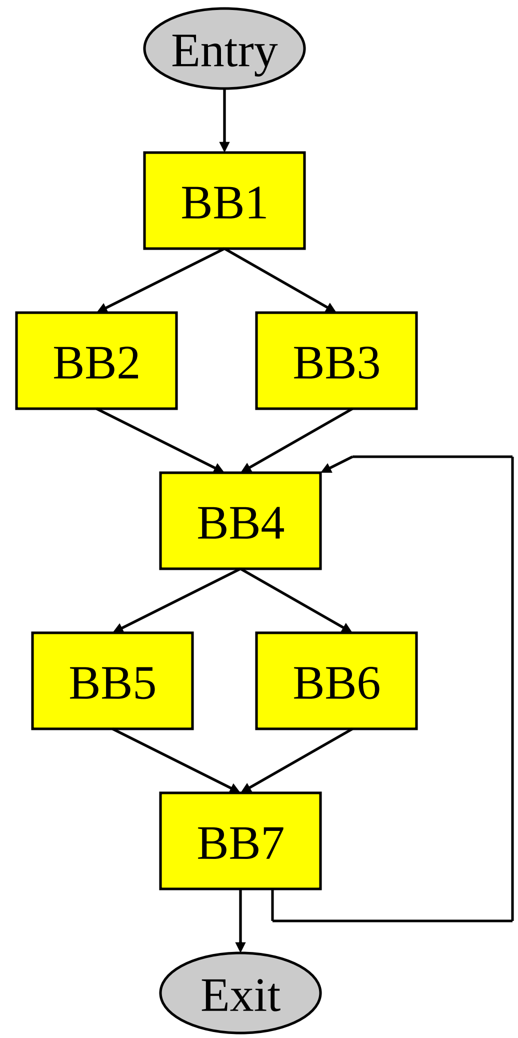


# Post Dominator Example 2



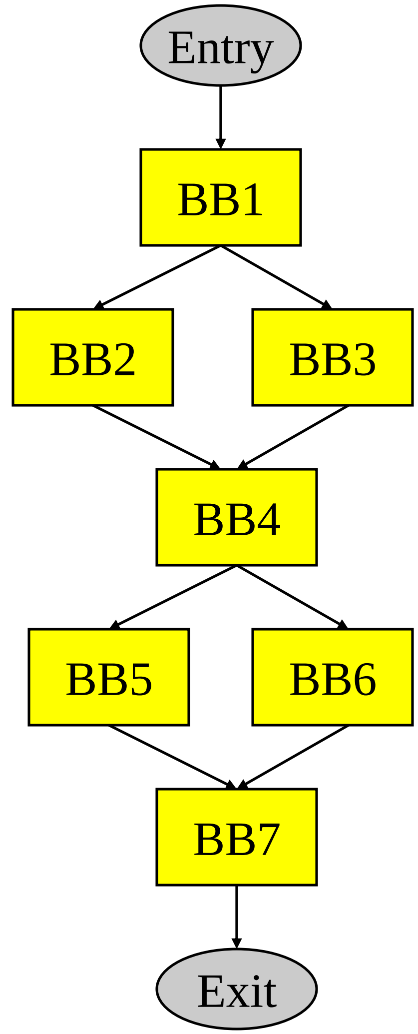
# Immediate Post Dominator

- Defn: Immediate post dominator (ipdom) – Each node n has a unique immediate post dominator m that is the first post dominator of n on any path from n to the Exit
  - » Closest node that post dominates
  - » First breadth-first successor that post dominates a node



# Why Do We Care About Dominators?

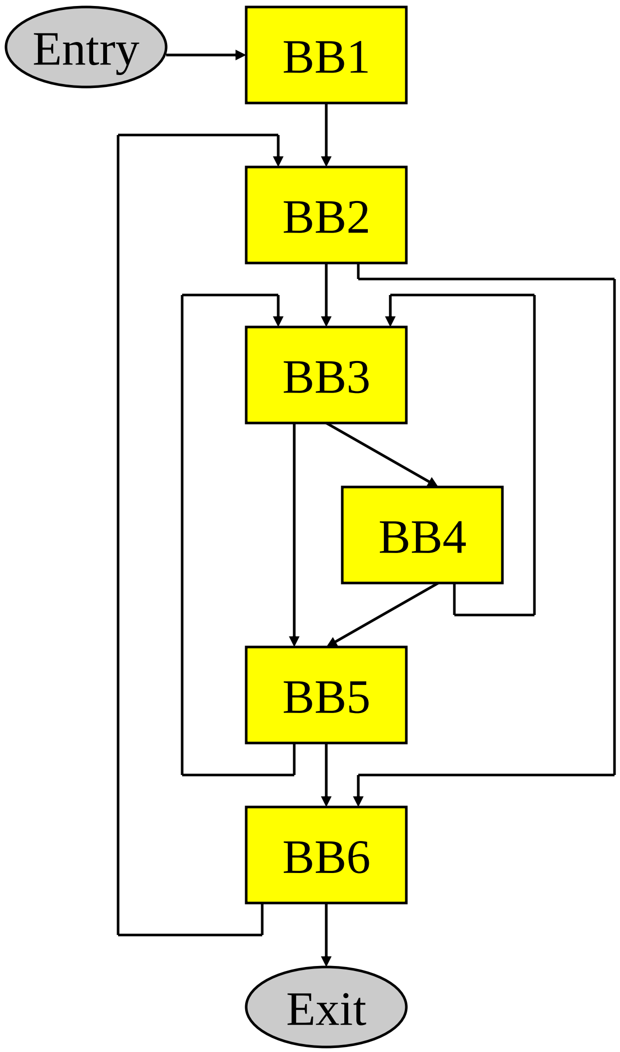
- Loop detection – next subject
- Dominator
  - Guaranteed to execute before
  - Redundant computation – an op is redundant if it is computed in a dominating BB
  - Most global optimizations use dominance info
- Post dominator
  - Guaranteed to execute after
  - Make a guess (ie 2 pointers do not point to the same locn)
  - Check they really do not point to one another in the post dominating BB



# Natural Loops

- Cycle suitable for optimization
  - Discuss optimizations later
- 2 properties
  - Single entry point called the <u>header</u>
    - Header <u>dominates</u> all blocks in the loop
  - Must be one way to iterate the loop (ie at least 1 path back to the header from within the loop) called a <u>backedge</u>
- Backedge detection
  - Edge, x→ y where the target (y) dominates the source (x)

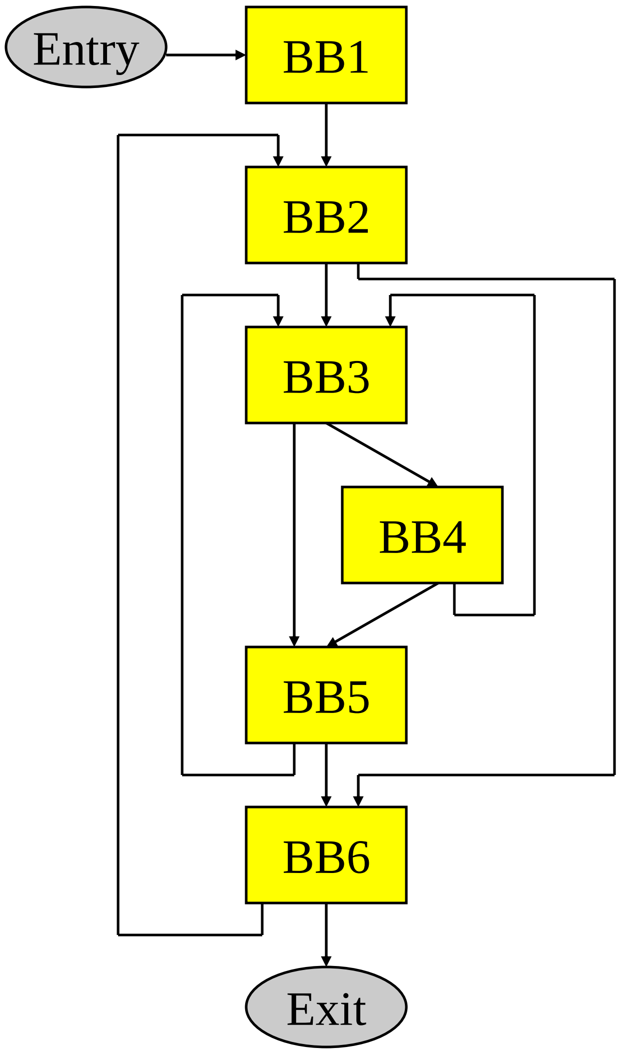
# Backedge Example



# Loop Detection

- Identify all backedges using Dom info
- Each backedge (x → y) defines a loop
  - Loop header is the backedge target (y)
  - Loop BB – basic blocks that comprise the loop
    - All predecessor blocks of x for which control can reach x without going through y are in the loop
- Merge loops with the same header
  - I.e., a loop with 2 continues
  - LoopBackedge = LoopBackedge1 + LoopBackedge2
  - LoopBB = LoopBB1 + LoopBB2
- Important property
  - Header dominates all LoopBB

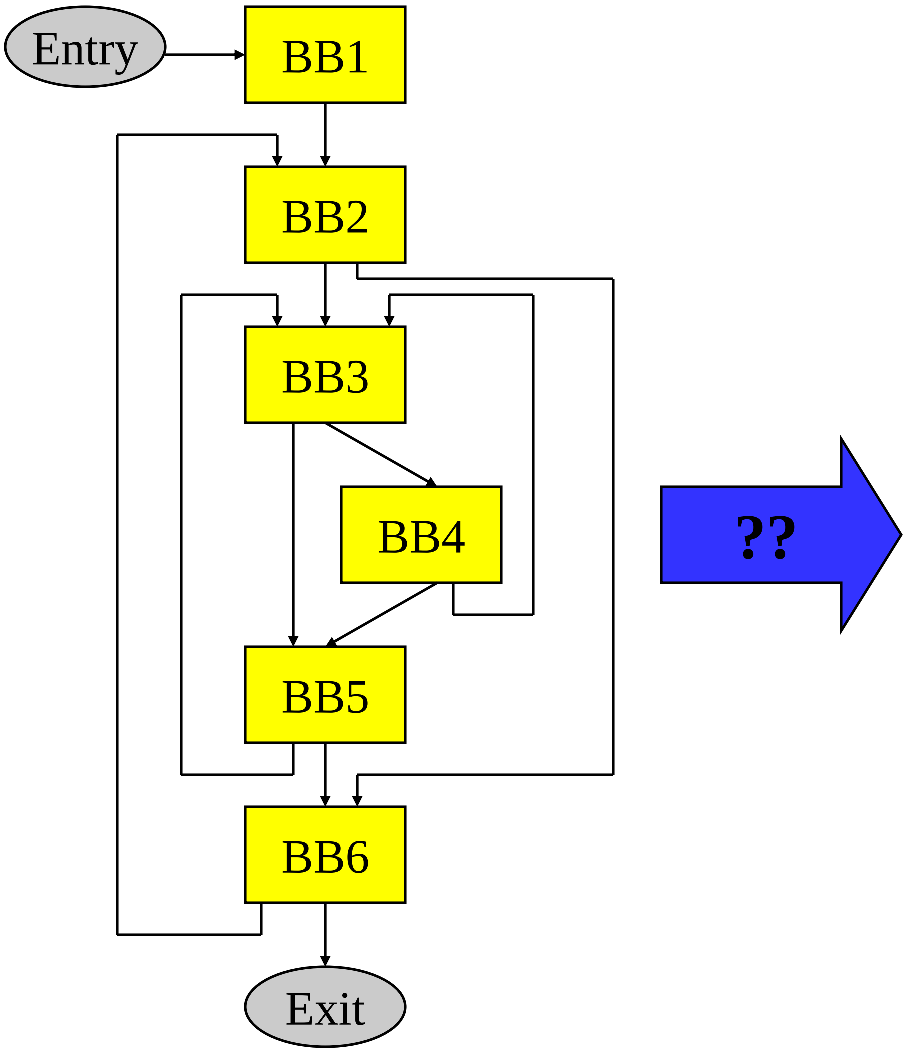
# Loop Detection Example



# Important Parts of a Loop

- Header, LoopBB
- Backedges, BackedgeBB
- Exitedges, ExitBB
  - For each LoopBB, examine each outgoing edge
  - If the edge is to a BB not in LoopBB, then its an exit
- Preheader (Preloop)
  - New block before the header (falls through to header)
  - Whenever you invoke the loop, preheader executed
  - Whenever you iterate the loop, preheader NOT executed
  - All edges entering header
    - Backedges – no change
    - All others, retarget to preheader
- Postheader (Postloop) - analogous

# Find the Preheaders for each Loop

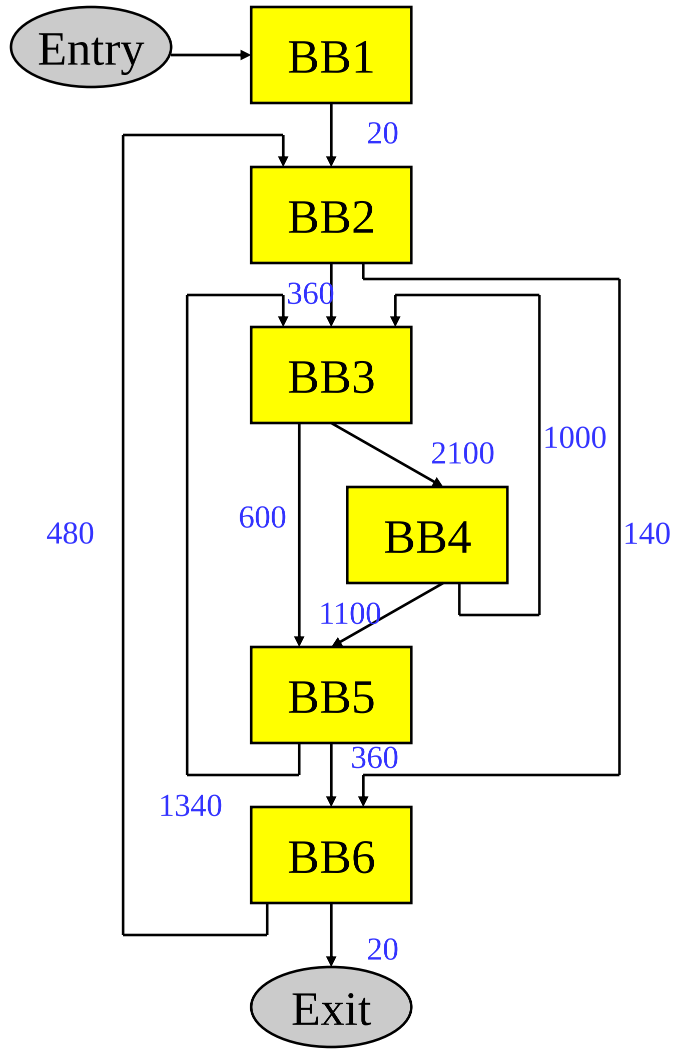


# Characteristics of a Loop

- Nesting (generally within a function scope)
  - Inner loop – Loop with no loops contained within it
  - Outer loop – Loop contained within no other loops
  - Nesting depth
    - depth(outer loop) = 1
    - depth = depth(parent or containing loop) + 1
- Trip count (average trip count)
  - How many times (on average) does the loop iterate
  - for (I=0; I<100; I++) → trip count = 100
  - With profile info:
    - Ave trip count = weight(header) / weight(preheader)

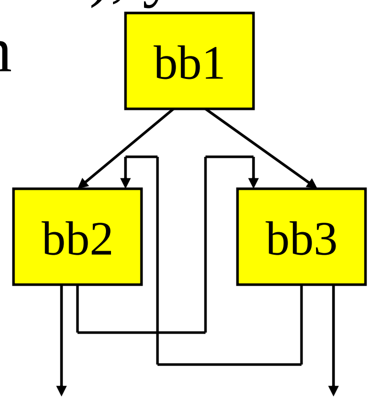
# Trip Count Calculation Example

Calculate the trip counts for all the loops in the graph



# Reducible Flow Graphs

- A flow graph is reducible if and only if we can partition the edges into 2 disjoint groups often called forward and back edges with the following properties
  - » The forward edges form an acyclic graph in which every node can be reached from the Entry
  - » The back edges consist only of edges whose destinations dominate their sources
- More simply – Take a CFG, remove all the backedges (x→ y where y dominates x), you should have a connected, acyclic graph




Non-reducible!