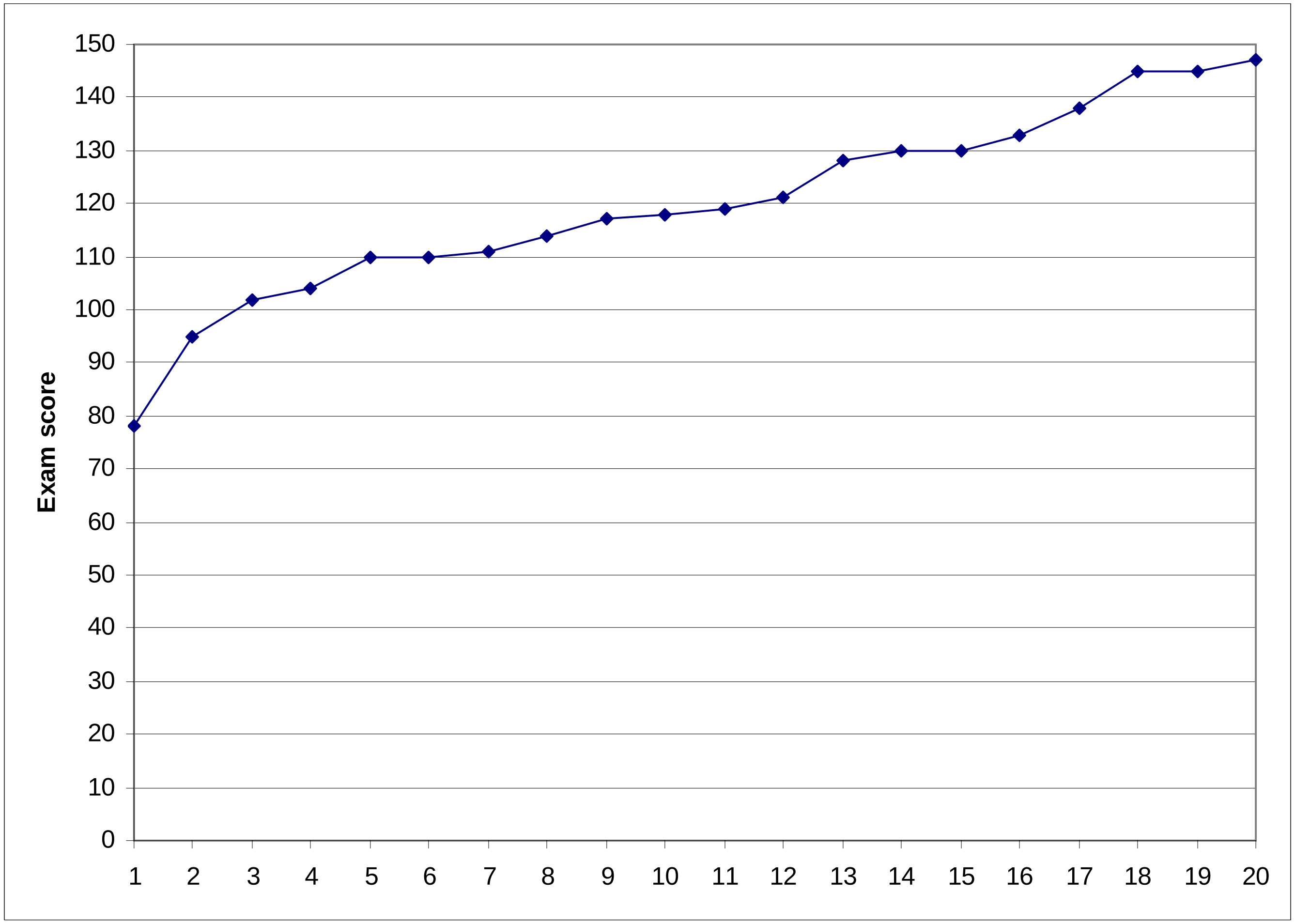
Control Flow 1:
Control Flow Graph, Dominators

EECS 483 – Lecture 19
University of Michigan
Monday, November 13, 2006

Exam 1 Results

Average: 119 Stdev: 17.8 High: 147



From Last Time: Memory Alignment

- Cannot arbitrarily pack variables into memory → Need to worry about alignment
- Golden rule – Address of a variable is aligned based on the size of the variable
  - » Char is byte aligned (any addr is fine)
  - » Short is halfword aligned (LSB of addr must be 0)
  - » Int is word aligned (2 LSBs of addr must be 0)
  - » This rule is for C/C++, other languages may have a slightly different rules

From Last Time: Structure Alignment

- Each field is layed out in the order it is declared using Golden Rule for aligning
- Identify largest field
  - » Starting address of overall struct is aligned based on the largest field
  - » Size of overall struct is a multiple of the largest field
  - » Reason for this is so can have an array of structs

From Last Time: Class Problem

How many bytes of memory does the following sequence of C declarations require (int = 4 bytes) ? Assume we start at a word aligned address, say 1000

short a[100];	size = 200, halfword aligned, maps to addrs 1000 - 1199
char b;	size = 1, byte aligned, maps to addr 1200
int c;	size = 4, word aligned, maps to addrs 1204-1207
double d;	size = 8, double aligned, maps to addrs, 1208-1215
short e;	size = 2, halfword aligned, maps to addrs, 1216-1217
struct {	max field = int, thus must be word aligned, start at addr 1220
char f;	size = 1, byte aligned, maps to addr, 1220
int g[1];	size = 4, word aligned, maps to addrs, 1224-1227
char h[2];	size = 2, byte aligned, maps to addrs 1228-1229
} i;	overall size of struct must be multiple of 4, thus pad out to1231

Total size = 232 bytes

Reading

- Generally over the next few weeks we will focus on Chs 9/10 of the Red Dragon book
- Today's class material:
  - » 9.4
  - » 10.1, 10.4

Compiler Backend Introduction

- Work at the assembly level
- 2 major concerns
  - How to make the code go faster
    - Machine independent opti
    - Machine dependent opti
    - Analyze program, understand its behavior, then transform it to a more efficient form
  - Map program onto real hardware
    - Deal with limitations of processor
    - Virtual to physical binding (resource binding)
  - Code size is 3rd concern, but not that important

Compiler Backend Structure

Improve code quality (machine independent opti

- Control flow analysis / Control flow optimization — Branching structure
- Dataflow analysis / Dataflow optimization — Computation instructions

Virtual to physical mapping and machine dependent opti

- Instruction Selection — Bind instrs to physical realizations
- Instruction Scheduling — Bind instrs to physical resources
- Register Allocation — Bind virtual regs to physical regs
- Machine Code Emission/Opti

Compiler Backend IR

- Low Level IR (intermediate representation)
  - Machine independent assembly code
    - Instruction set for abstract machine
  - r1 = r2 + r3 or equivalently add r1, r2, r3
    - Opcode
    - Operands
      - Virtual registers – infinite number of these
      - Special registers – stack pointer, pc, etc.
      - Literals – compile-time constants (no limit on size of these)
      - Symbolic names – start of array, branch targets

Control Flow

- Control transfer = branch (taken or fall-through)
- Control flow
  - » Branching behavior of an application
  - » What sequences of instructions can be executed
- Execution → Dynamic control flow
  - » Direction of a particular instance of a branch
  - » Predict, speculate, squash, etc.
- Compiler → Static control flow
  - » Not executing the program
  - » Input not known, so what could happen, worst case

Basic Block (BB)

- Group operations into units with equivalent execution conditions
- Defn: Basic block – a sequence of consecutive operations in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end
  - » Straight-line sequence of instructions
  - » If one operation is executed in a BB, they all are
- Finding BB's
  - » The first operation starts a BB
  - » Any operation that is the target of a branch starts a BB
  - » Any operation that immediately follows a branch starts a BB

Class Problem

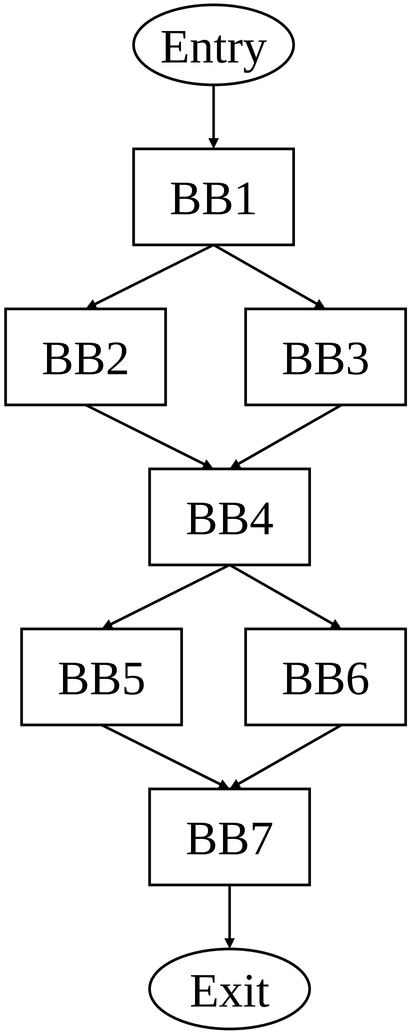
Identify the BBs in this code sequence:

Remember: 2 main rules:
* Rule 1: Each branch ends a basic block
* Rule 2: Each branch target starts a basic block

```
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 L2
L7: beq r2 100 L10
L8: r5 = r9 + 1
L9: r7 = r7 & 3
L10: r9 = load (r3)
L11: store(r9, r1)
```

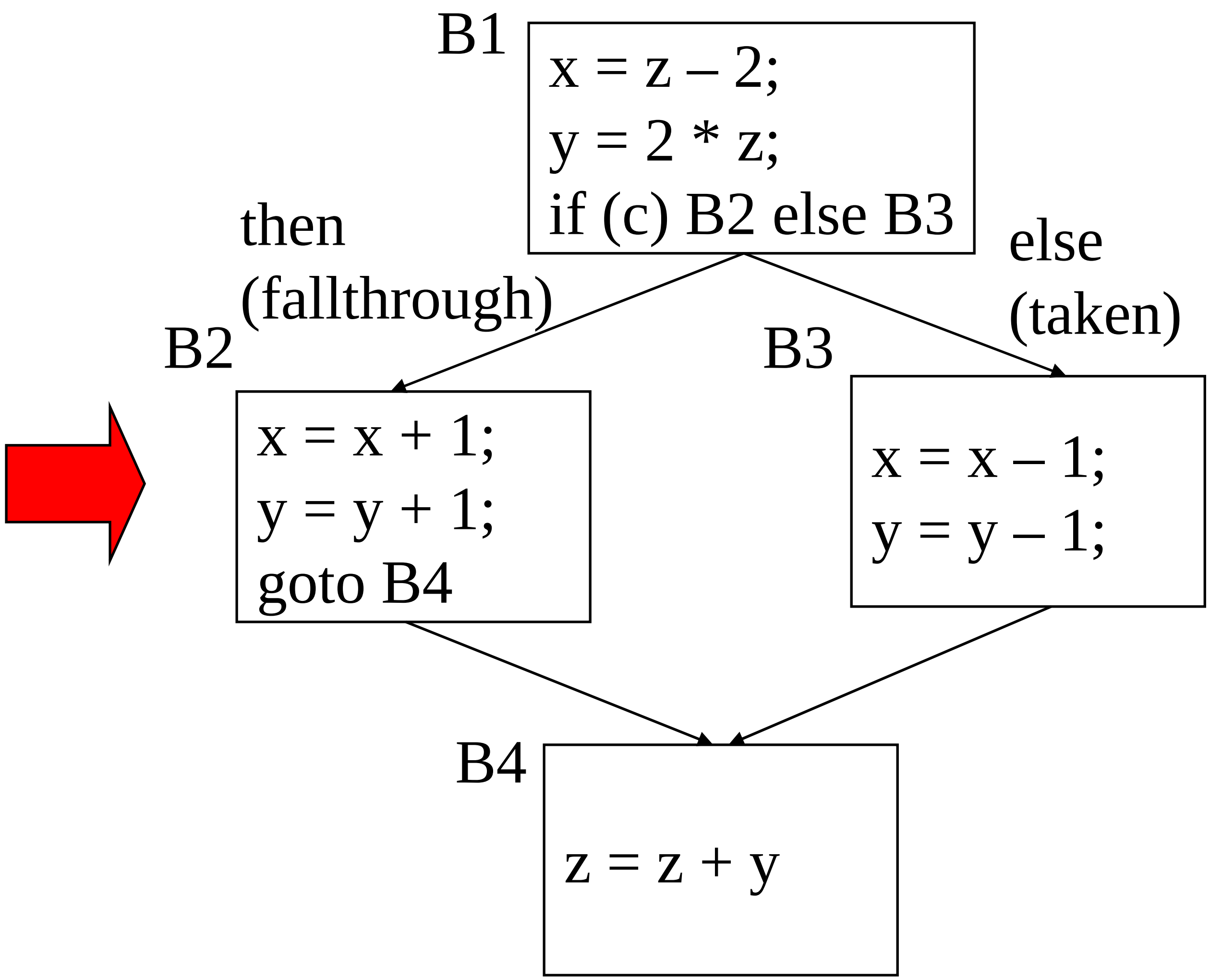
Control Flow Graph (CFG)

- 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



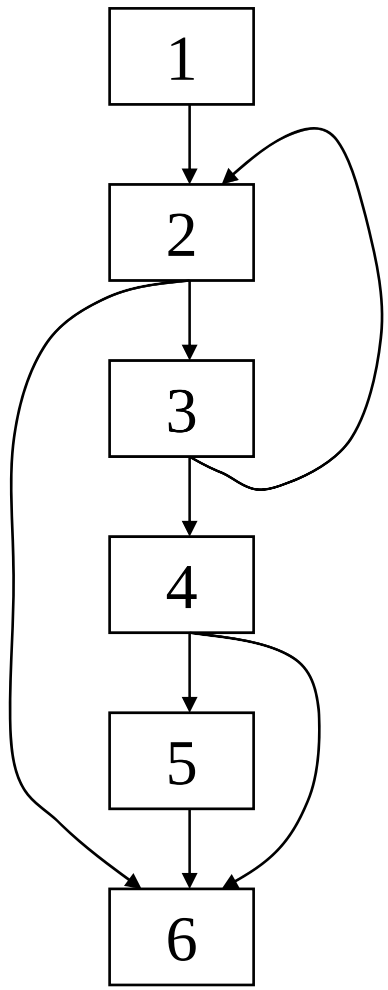
CFG Example

```
x = z – 2;
y = 2 * z;
if (c) {
    x = x + 1;
    y = y + 1;
}
else {
    x = x – 1;
    y = y – 1;
}
z = x + y
```



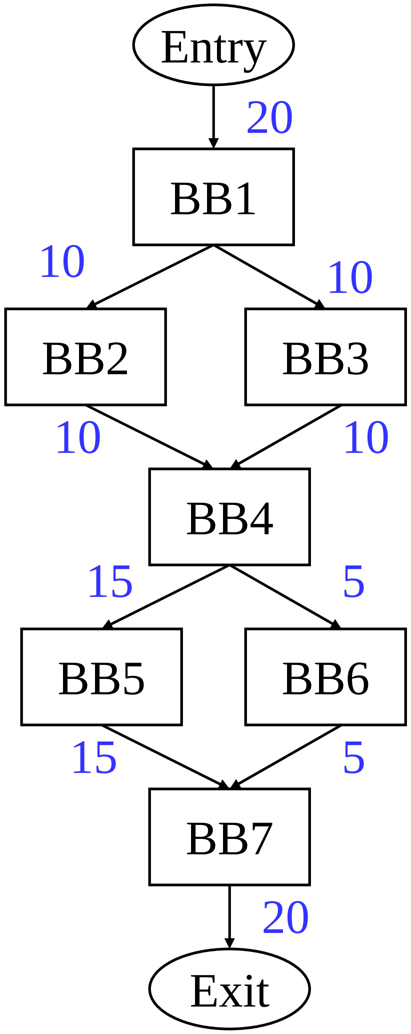
Another CFG Example

1	L1: r7 = load(r8)
2	L2: r1 = r2 + r3 L3: beq r1, 0, L10
3	L4: r4 = r5 * r6 L5: r1 = r1 + 1 L6: beq r1 100 L2
4	L7: beq r2 100 L10
5	L8: r5 = r9 + 1 L9: r7 = r7 & 3
6	L10: r9 = load (r3) L11: store(r9, r1)



Weighted CFG

- Profiling – Run the application on 1 or more sample inputs, record some behavior
  - Control flow profiling**
    - edge profile
    - block profile
  - Path profiling
- Annotate control flow profile onto a CFG → weighted CFG
- Optimize more effectively with profile info!!
  - Optimize for the common case
  - Make educated guess



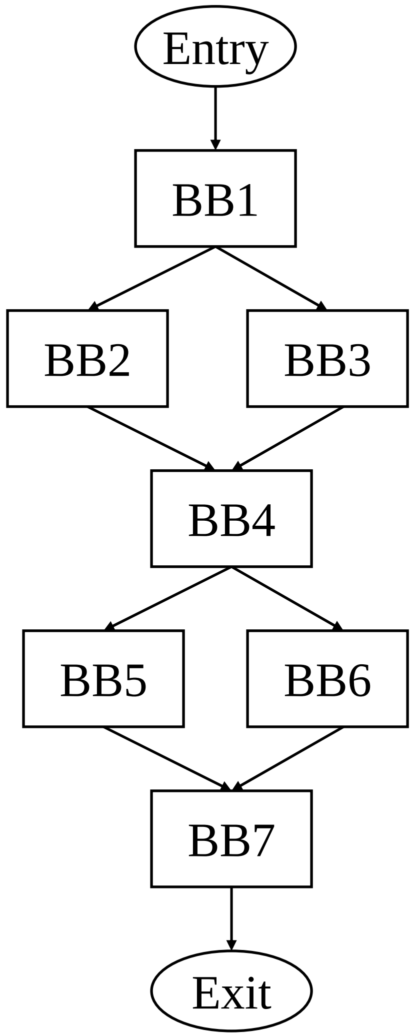
Control Flow Analysis

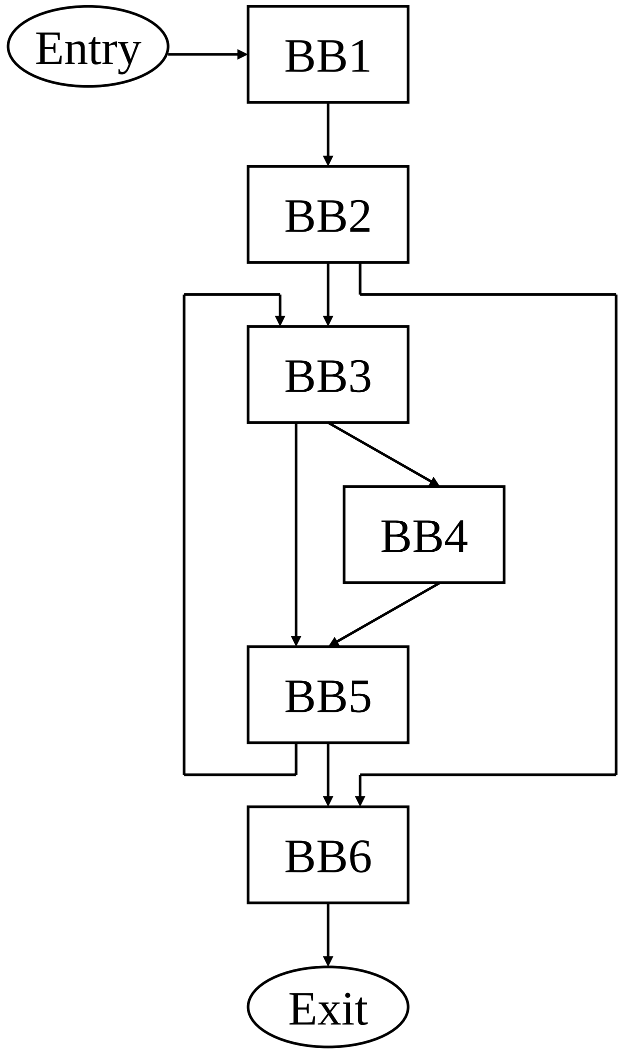
- Determining properties of the program branch structure
  - Static properties → Not executing the code
  - Properties that exist regardless of the run-time branch directions
  - Use CFG
  - Optimize efficiency of control flow structure
- Determine instruction execution properties
  - Global optimization of computation operations
  - Discuss this later

Dominator

- 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

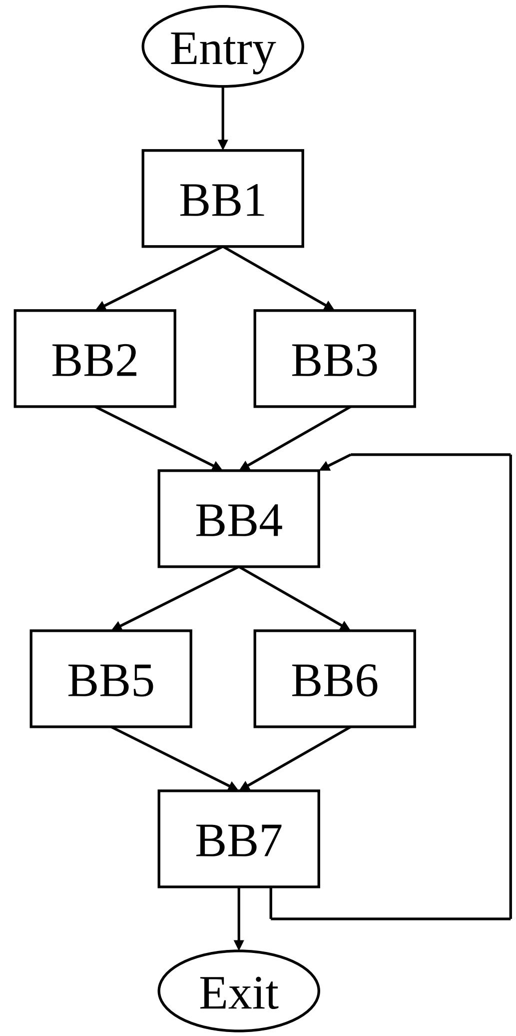
Dominator Examples





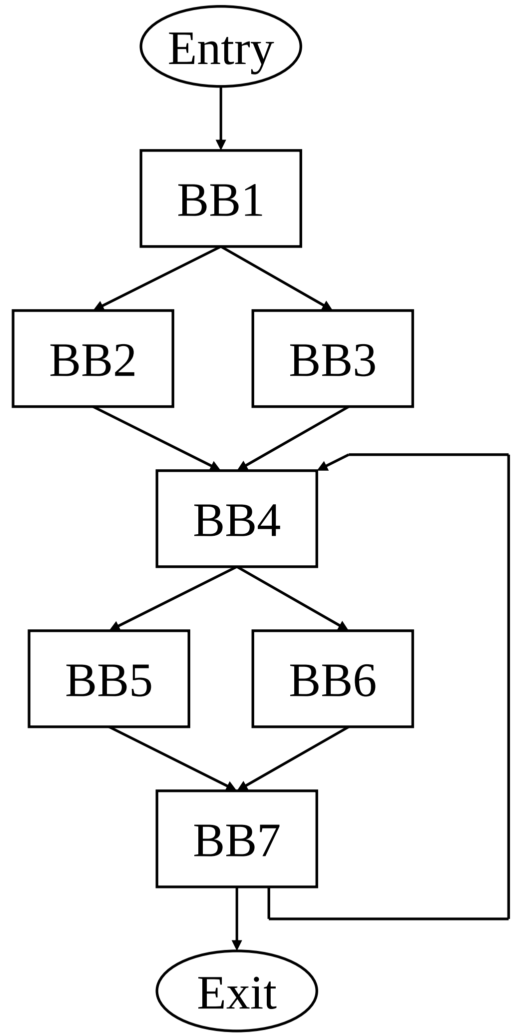
Dominator Analysis

- 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

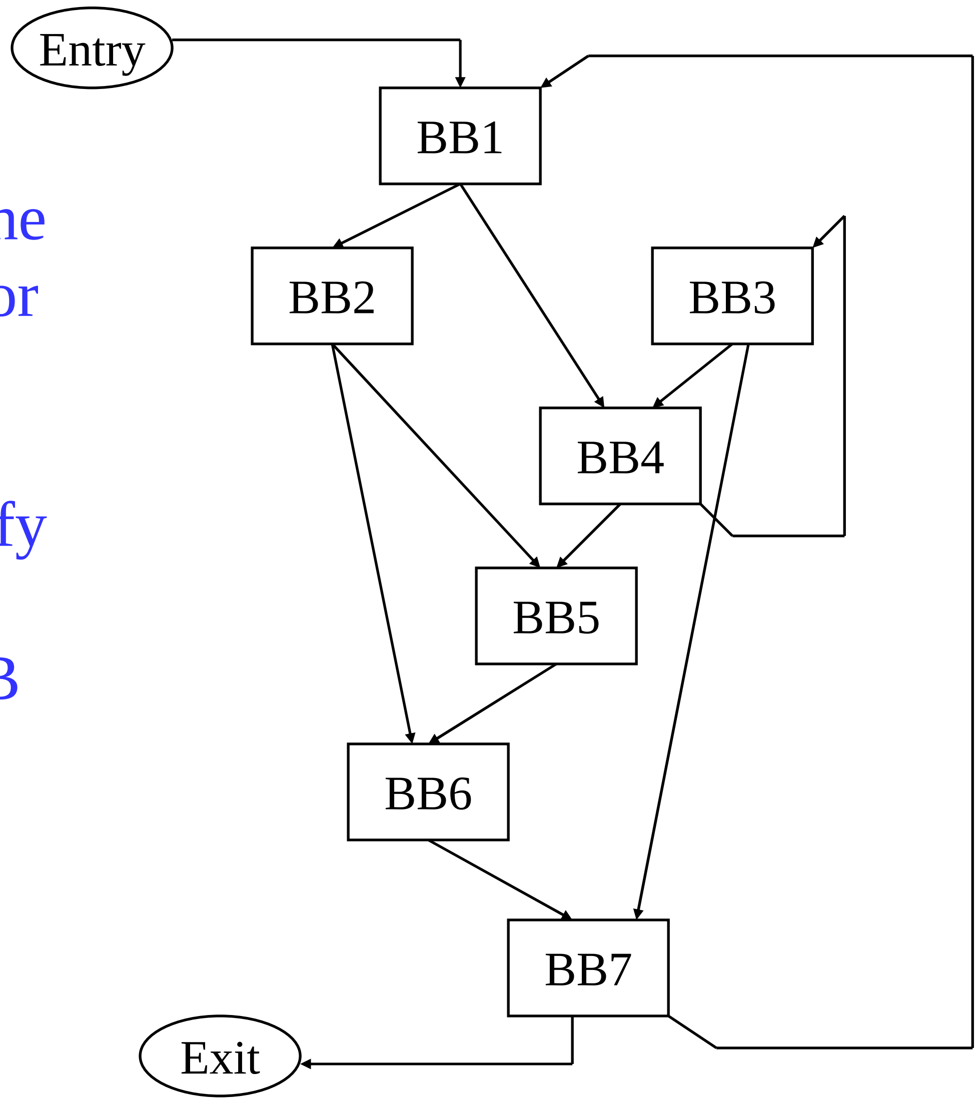
- Defn: Immediate dominator (idom)– Each node n has a unique immediate dominator m that is the last dominator of n on any path from the initial node to n
  - » Closest node that dominates



Class Problem

Calculate the DOM set for each BB

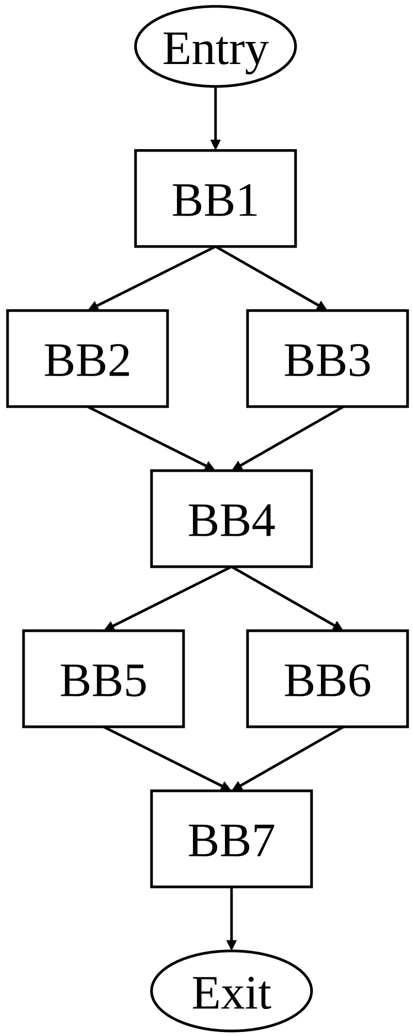
Also identify the iDOM for each BB

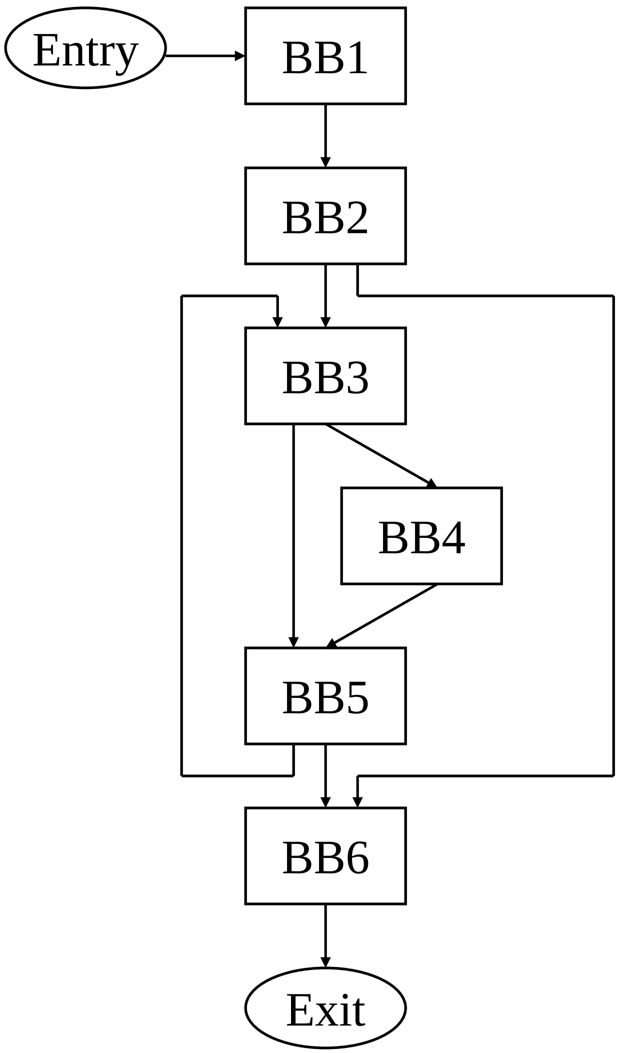


Post Dominator

- 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

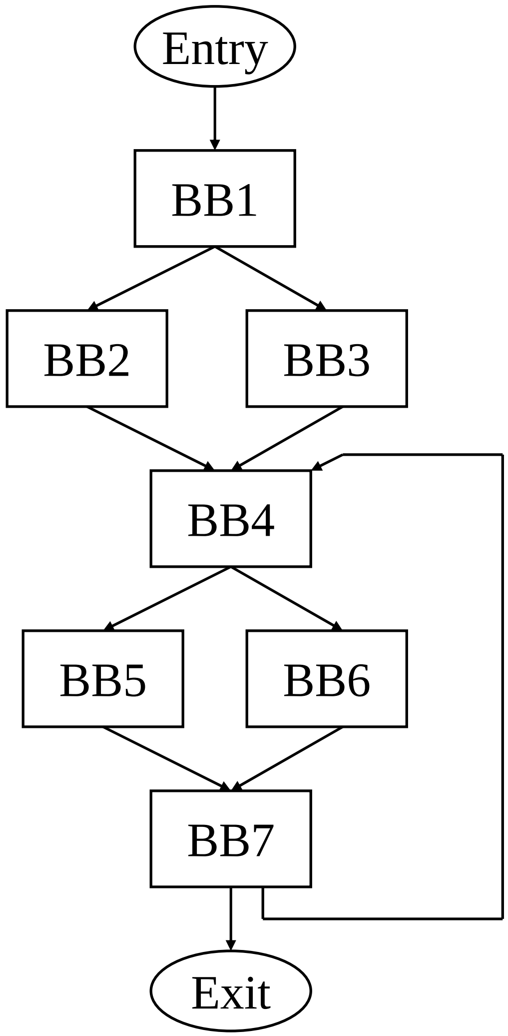
Post Dominator Examples





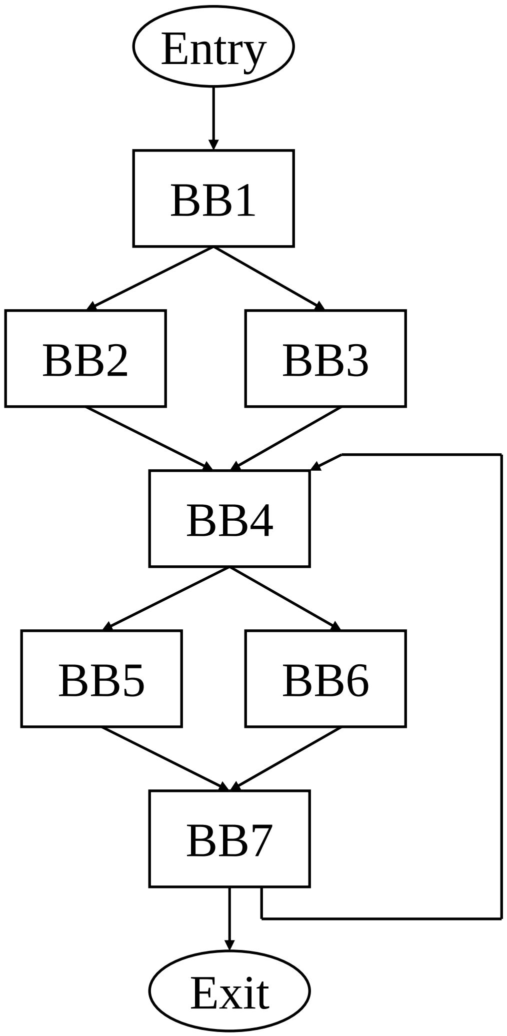
Post Dominator Analysis

- Compute 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



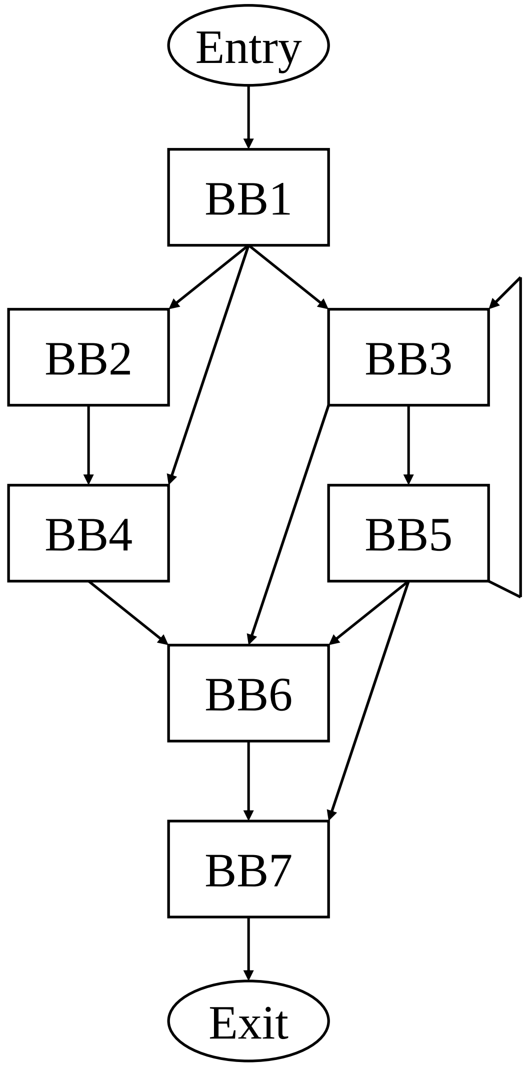
Immediate Post Dominator

- 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



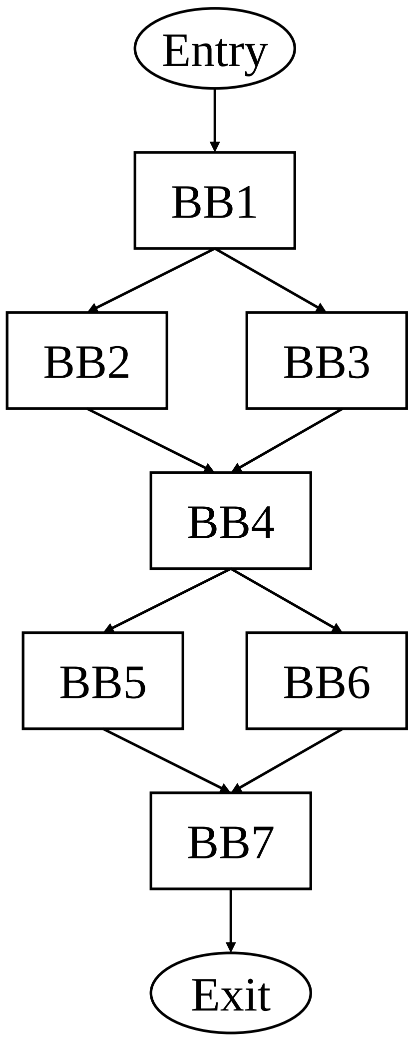
Class Problem

Calculate the PDOM set for each BB



Why Do We Care About Dominators?

- Loop detection – next subject
- Dominator
  - » Guaranteed to execute before
  - » Redundant computation – an op can only be 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 opti 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, $x \rightarrow y$ where the target (y) dominates the source (x)

Backedge Example

BE = target dominates source

E → 1 : No

1 → 2 : No

2 → 3 : No

2 → 6 : No

3 → 4 : No

3 → 5 : No

4 → 3 : Yes

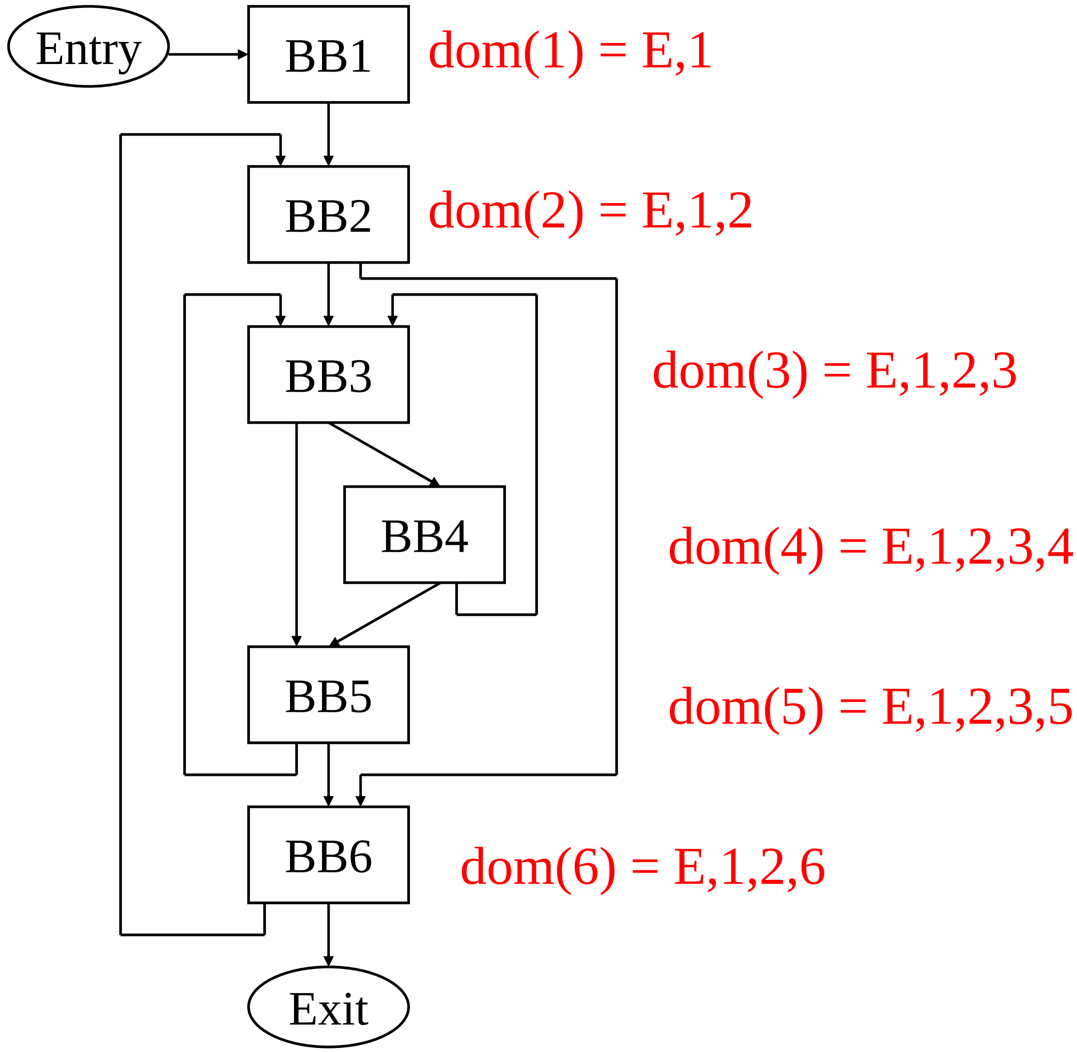
4 → 5 : No

5 → 3 : Yes

5 → 6 : No

6 → 2 : Yes

6 → X : No



In this example, BE = edge from higher BB to lower BB, not always this easy!