EECS 583 – Class 11
Instruction Scheduling

University of Michigan

October 12, 2011

Reading Material + Announcements

- Today's class
  - "The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors," P. Chang et al., IEEE Transactions on Computers, 1995, pp. 353-370.
- Next class
  - "Sentinel Scheduling for VLIW and Superscalar Processors", S. Mahlke et al., ASPLOS-5, Oct. 1992, pp.238-247.
- Reminder: HW 2 – Speculative LICM
  - Due Week from Fri → Get busy, go bug Daya if you are stuck!
- Class project proposals
  - Week of Oct 24: Daya and I will meet with each group to discuss informal project proposal
  - Signup sheet available next week
  - Think about partners/topic!

Homework Problem From Last Time - Answer

```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400) goto loop
```

Optimize the unrolled loop

Renaming
Tree height reduction
Ind/Acc expansion

```
loop:
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
r1 = load(r2)
r5 = r6 + 3
r6 = r5 + r1
r2 = r2 + 4
if (r2 < 400)
    goto loop
```

```
loop:
  r1 = load(r2)
  r5 = r1 + 3
  r6 = r6 + r5
  r2 = r2 + 4
  r11 = load(r2)
  r15 = r11 + 3
  r6 = r6 + r15
  r2 = r2 + 4
  r21 = load(r2)
  r25 = r21 + 3
  r6 = r6 + r25
  r2 = r2 + 4
  if (r2 < 400)
      goto loop
```

after renaming and tree height reduction

```
r16 = r26 = 0
loop:
 r1 = load(r2)
 r5 = r1 + 3
 r6 = r6 + r5
 r11 = load(r2+4)
 r15 = r11 + 3
 r16 = r16 + r15
 r21 = load(r2+8)
 r25 = r21 + 3
 r26 = r26 + r25
 r2 = r2 + 12
 if (r2 < 400)
     goto loop
 r6 = r6 + r16
 r6 = r6 + r26
```

after acc and ind expansion

From Last Time: Dependences

Register Dependences

Flow

r1 = r2 + r3

r4 = r1 * 6

Output

r1 = r2 + r3

r1 = r4 * 6

Anti

r1 = r2 + r3

r2 = r5 * 6

Memory Dependences

Mem-flow

store (r1, r2)

r3 = load(r1)

Mem-output

store (r1, r2)

store (r1, r3)

Mem-anti

r2 = load(r1)

store (r1, r3)

Control Dependences

Control (C1)

if (r1 != 0)

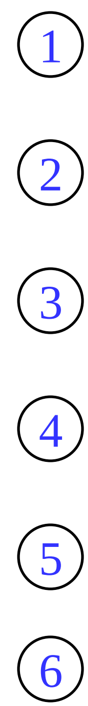
r2 = load(r1)

From Last Time: Dependence Graph

- Represent dependences between operations in a block via a DAG
  - Nodes = operations
  - Edges = dependences
- Single-pass traversal required to insert dependences
- Example

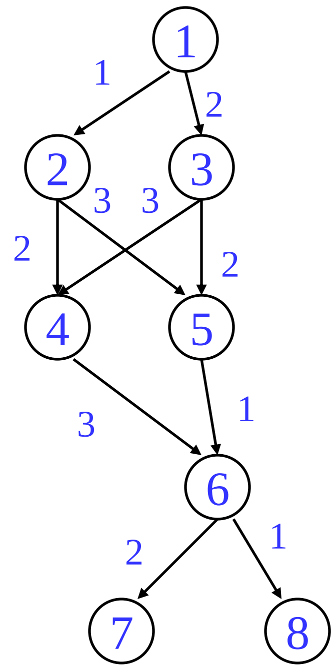
1: r1 = load(r2)
2: r2 = r1 + r4
3: store (r4, r2)
4: p1 = cmpp (r2 < 0)
5: branch if p1 to BB3
6: store (r1, r2)

BB3:



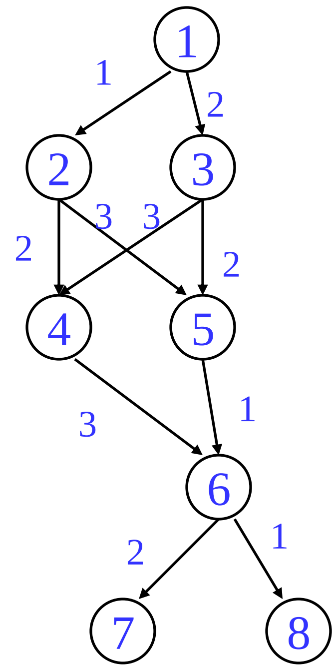
Dependence Graph Properties - Estart

- Estart = earliest start time, (as soon as possible - ASAP)
  - Schedule length with infinite resources (dependence height)
  - Estart = 0 if node has no predecessors
  - Estart = MAX(Estart(pred) + latency) for each predecessor node
  - Example



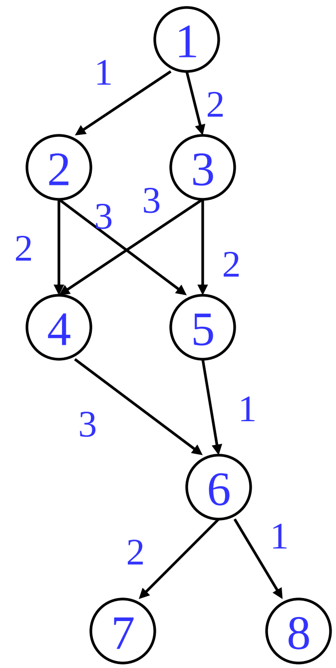
Lstart

- Lstart = latest start time, ALAP
  - Latest time a node can be scheduled s.t. sched length not increased beyond infinite resource schedule length
  - Lstart = Estart if node has no successors
  - Lstart = MIN(Lstart(succ) - latency) for each successor node
  - Example



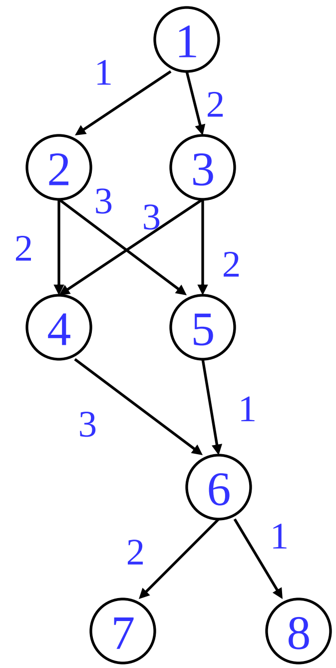
Slack

- Slack = measure of the scheduling freedom
  - Slack = Lstart – Estart for each node
  - Larger slack means more mobility
  - Example

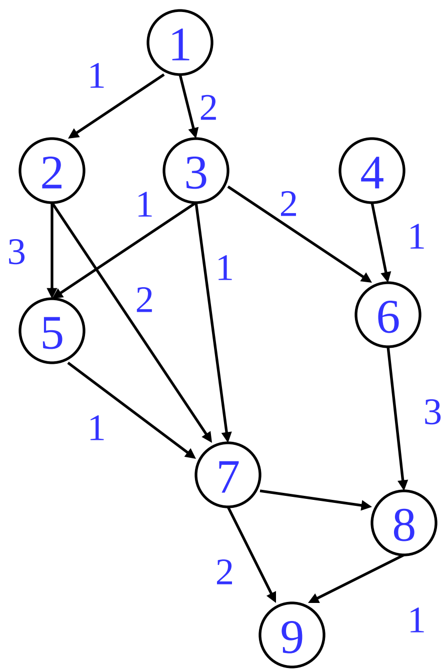


Critical Path

- Critical operations = Operations with slack = 0
  - » No mobility, cannot be delayed without extending the schedule length of the block
  - » Critical path = sequence of critical operations from node with no predecessors to exit node, can be multiple crit paths



Class Problem



Node	Estart	Lstart	Slack
1			
2			
3			
4			
5			
6			
7			
8			
9			

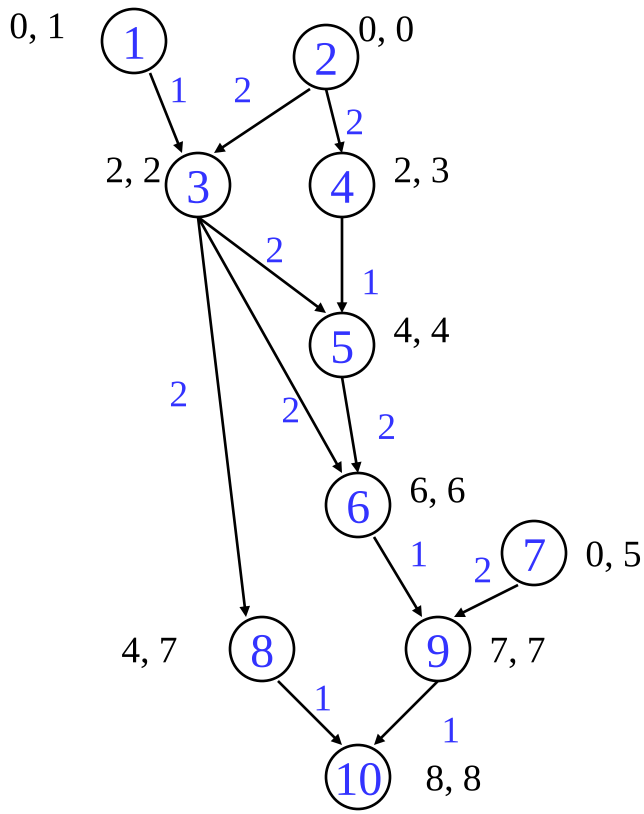
Critical path(s) =

Operation Priority

- Priority – Need a mechanism to decide which ops to schedule first (when you have multiple choices)
- Common priority functions
  - Height – Distance from exit node
    - Give priority to amount of work left to do
  - Slackness – inversely proportional to slack
    - Give priority to ops on the critical path
  - Register use – priority to nodes with more source operands and fewer destination operands
    - Reduces number of live registers
  - Uncover – high priority to nodes with many children
    - Frees up more nodes
  - Original order – when all else fails

Height-Based Priority

- Height-based is the most common
  - priority(op) = MaxLstart – Lstart(op) + 1

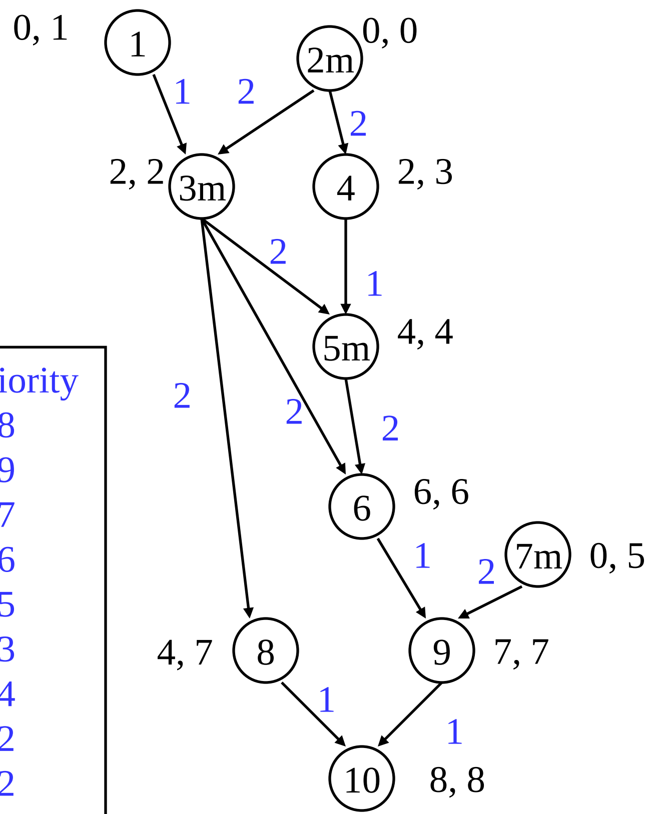


op	priority
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

List Scheduling (aka Cycle Scheduler)

- Build dependence graph, calculate priority
- Add all ops to UNSCHEDULED set
- time = -1
- while (UNSCHEDULED is not empty)
  - time++
  - READY = UNSCHEDULED ops whose incoming dependences have been satisfied
  - Sort READY using priority function
  - For each op in READY (highest to lowest priority)
    - op can be scheduled at current time? (are the resources free?)
      - Yes, schedule it, op.issue_time = time
        - Mark resources busy in RU_map relative to issue time
        - Remove op from UNSCHEDULED/READY sets
      - No, continue

Cycle Scheduling Example



op	priority
1	8
2	9
3	7
4	6
5	5
6	3
7	4
8	2
9	2
10	1

RU_map

time	ALU	MEM
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Schedule

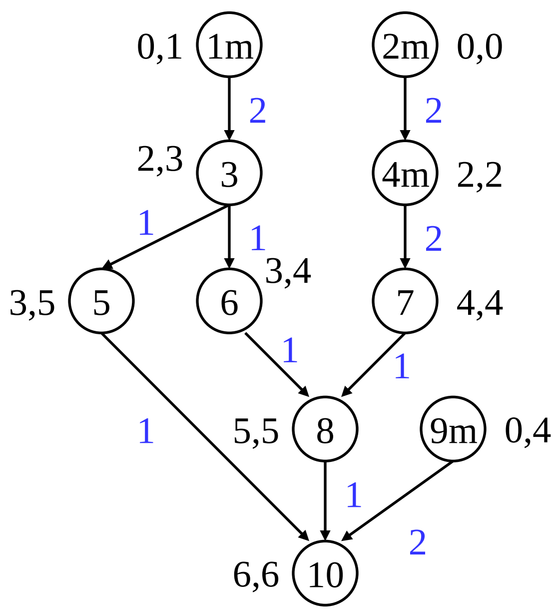
time	Ready	Placed
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

List Scheduling (Operation Scheduler)

- Build dependence graph, calculate priority
- Add all ops to UNSCHEDULED set
- while (UNSCHEDULED not empty)
  - op = operation in UNSCHEDULED with highest priority
  - For time = estart to some deadline
    - Op can be scheduled at current time? (are resources free?)
      - Yes, schedule it, op.issue_time = time
        - Mark resources busy in RU_map relative to issue time
        - Remove op from UNSCHEDULED
      - No, continue
  - Deadline reached w/o scheduling op? (could not be scheduled)
    - Yes, unplace all conflicting ops at op.estart, add them to UNSCHEDULED
    - Schedule op at estart
      - Mark resources busy in RU_map relative to issue time
      - Remove op from UNSCHEDULED

Homework Problem – Operation Scheduling

Machine: 2 issue, 1 memory port, 1 ALU
Memory port = 2 cycles, pipelined
ALU = 1 cycle



RU_map

time	ALU	MEM
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

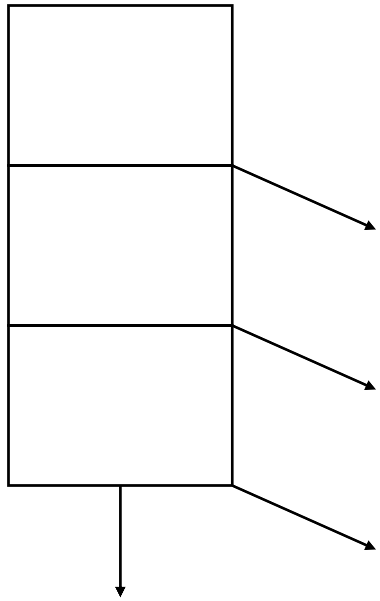
Schedule

time	Ready	Placed
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

1. Calculate height-based priorities
2. Schedule using <u>Operation</u> scheduler

Generalize Beyond a Basic Block

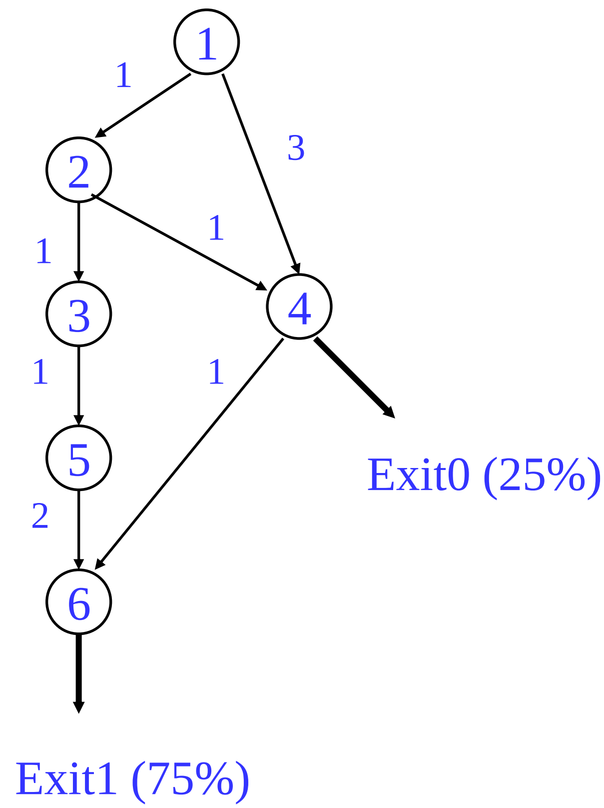
- Superblock
  - Single entry
  - Multiple exits (side exits)
  - No side entries
- Schedule just like a BB
  - Priority calculations needs change
  - Dealing with control deps



Lstart in a Superblock

- Not a single Lstart any more
  - 1 per exit branch (Lstart is a vector!)
  - Exit branches have probabilities

op	Estart	Lstart0	Lstart1
1			
2			
3			
4			
5			
6			



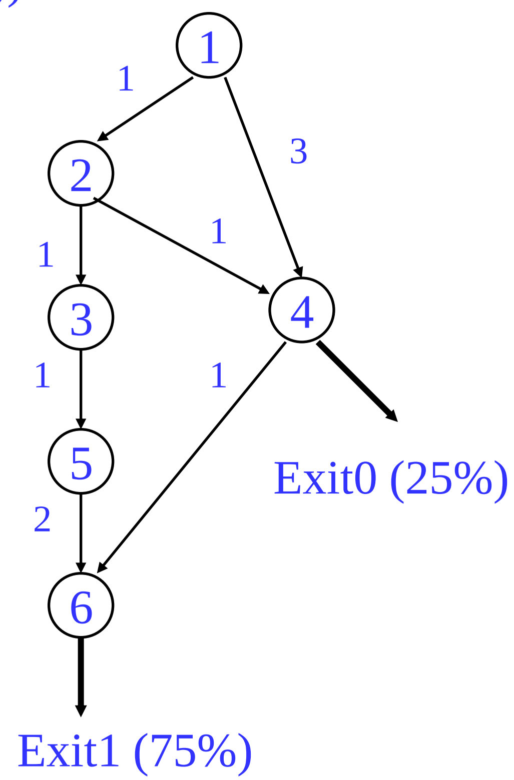
Operation Priority in a Superblock

- Priority – Dependence height and speculative yield
  - » Height from op to exit * probability of exit
  - » Sum up across all exits in the superblock

Priority(op) = SUM(Probi * (MAX_Lstart – Lstarti(op) + 1))

valid late times for op

op	Lstart0	Lstart1	Priority
1			
2			
3			
4			
5			
6			

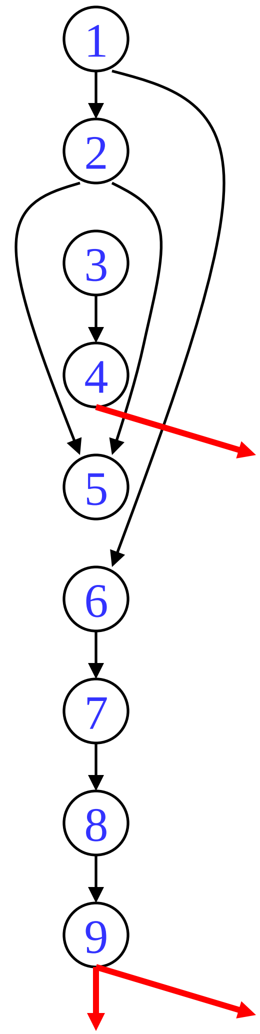


Dependences in a Superblock

Superblock

```
1: r1 = r2 + r3
2: r4 = load(r1)
3: p1 = cmpp(r3 == 0)
4: branch p1 Exit1
5: store (r4, -1)
6: r2 = r2 – 4
7: r5 = load(r2)
8: p2 = cmpp(r5 > 9)
9: branch p2 Exit2
```

Note: Control flow in red bold



* Data dependences shown, all are reg flow except 1→ 6 is reg anti

* Dependences define precedence ordering of operations to ensure correct execution semantics

* What about control dependences?

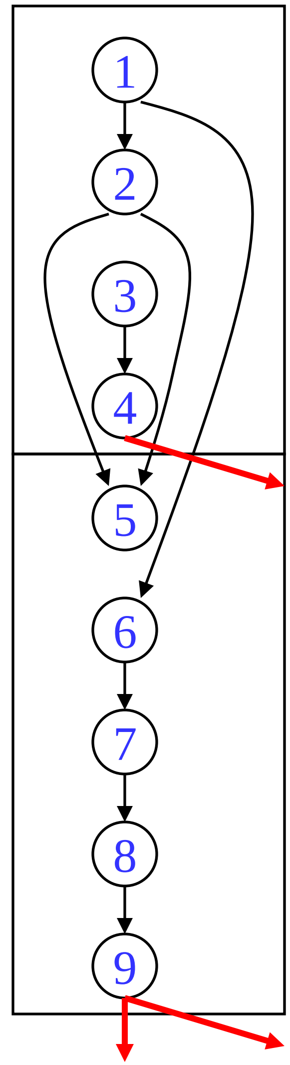
* Control dependences define precedence of ops with respect to branches

Conservative Approach to Control Dependences

Superblock

```
1: r1 = r2 + r3
2: r4 = load(r1)
3: p1 = cmpp(r3 == 0)
4: branch p1 Exit1
5: store (r4, -1)
6: r2 = r2 – 4
7: r5 = load(r2)
8: p2 = cmpp(r5 > 9)
9: branch p2 Exit2
```

Note: Control flow in red bold



* Make branches barriers, nothing moves above or below branches

* Schedule each BB in SB separately

* Sequential schedules

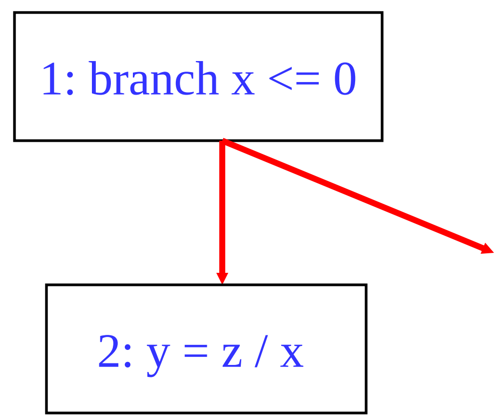
* Whole purpose of a superblock is lost

Upward Code Motion Across Branches

- Restriction 1a (register op)
  - The destination of op is not in liveout(br)
  - Wrongly kill a live value
- Restriction 1b (memory op)
  - Op does not modify the memory
  - Actually live memory is what matters, but that is often too hard to determine
- Restriction 2
  - Op must not cause an exception that may terminate the program execution when br is taken
  - Op is executed more often than it is supposed to (<u>speculated</u>)
  - Page fault or cache miss are ok
- Insert control dep when either restriction is violated

```
…
if (x > 0)
   y = z / x
…
```

control flow graph



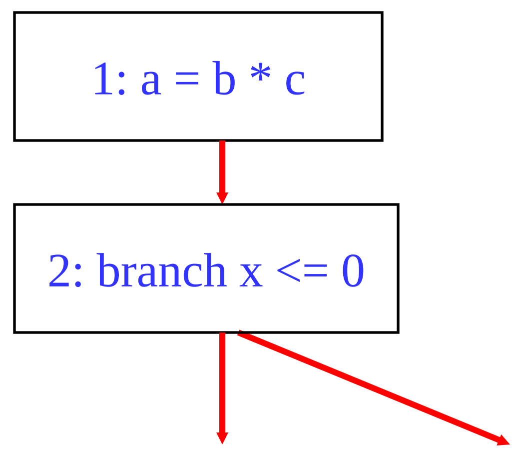
Downward Code Motion Across Branches

- Restriction 1 (liveness)
  - » If no compensation code
    - Same restriction as before, destination of op is not liveout
  - » Else, no restrictions
    - Duplicate operation along both directions of branch if destination is liveout
- Restriction 2 (speculation)
  - » Not applicable, downward motion is not speculation
- Again, insert control dep when the restrictions are violated
- Part of the philosphy of superblocks is no compensation code inseration hence R1 is enforced!

```
…
a = b * c
if (x > 0)

else
…
```

control flow graph



Add Control Dependences to a Superblock

Superblock

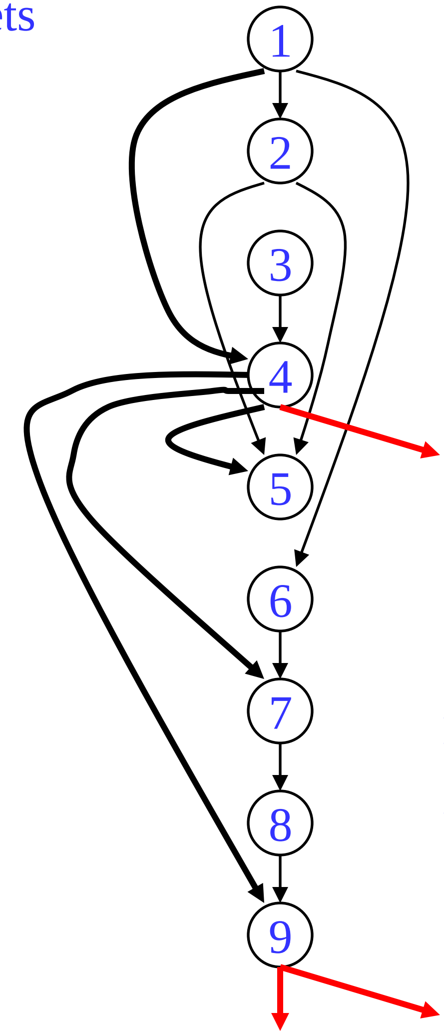
1: r1 = r2 + r3
2: r4 = load(r1)
3: p1 = cmpp(r2 == 0)
4: branch p1 Exit1
5: store (r4, -1)
6: r2 = r2 – 4
7: r5 = load(r2)
8: p2 = cmpp(r5 > 9)
9: branch p2 Exit2

Assumed liveout sets

{r1}

{r2}

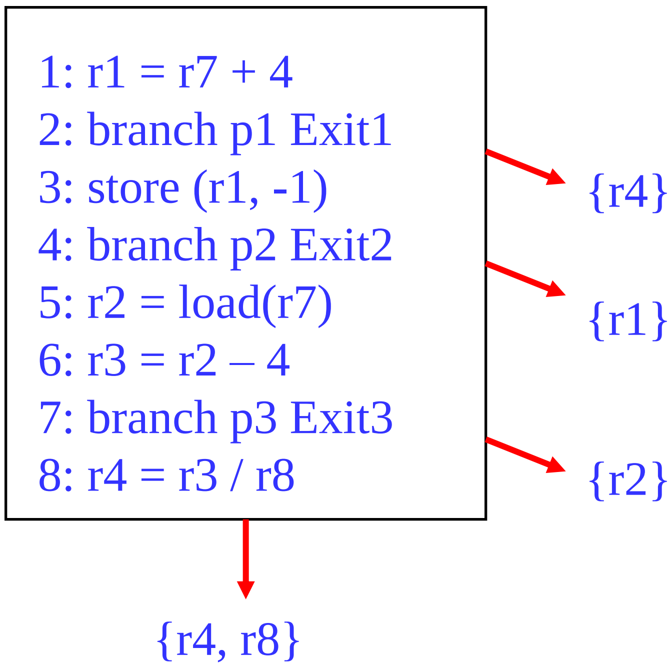
{r5}



All ops have cdep to op 9!

Notes: All branches are control dependent on one another.
If no compensation, all ops dependent on last branch

Class Problem



Draw the dependence graph