EECS 583 – Class 13
Superblock Scheduling

University of Michigan

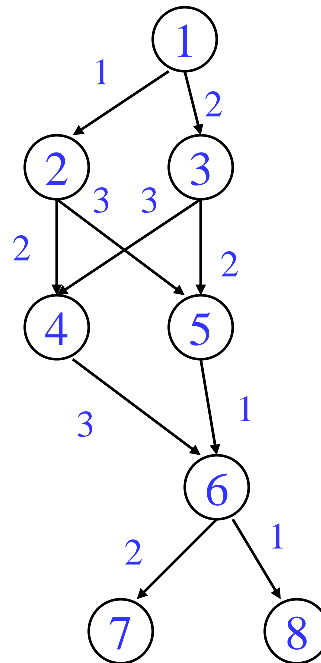
October 28, 2019

Announcements & Reading Material

- Project proposals
  - Due Wednesday, Oct 30, 11:59pm
  - 1 paragraph summary of what you plan to work on
    - Topic, what are you going to do, what is the goal, 1-2 references
  - Email to me & Sung & Armand, cc all your group members
- Midterm exam
  - Originally scheduled for Wed Nov 6
  - Moved to Wed Nov 13 in class. Likely in this room and another (or 2) so stay tuned for your room assignment.
  - More on review session and the content later. Prior exams will be posted!
- 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
  - "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops", B. Rau, MICRO-27, 1994, pp. 63-74.

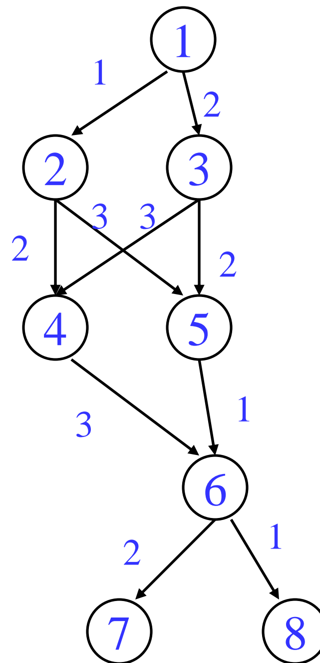
From Last Time: Dependence Graph Properties - Estart

- 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



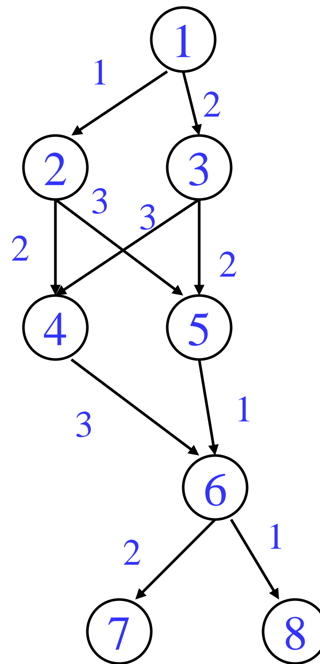
From Last Time: Lstart

- 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



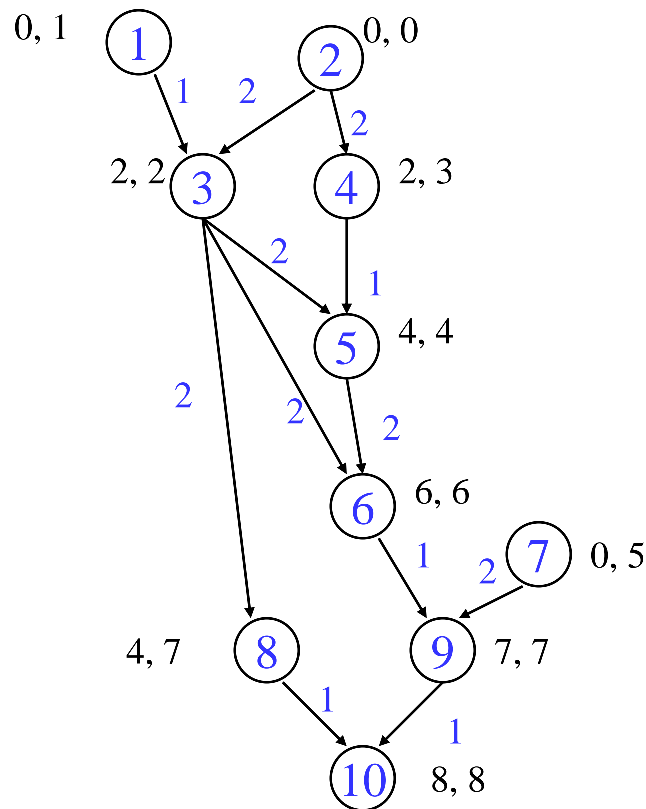
From Last Time: Critical Path

- 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



From Last Time: Height-Based Priority

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

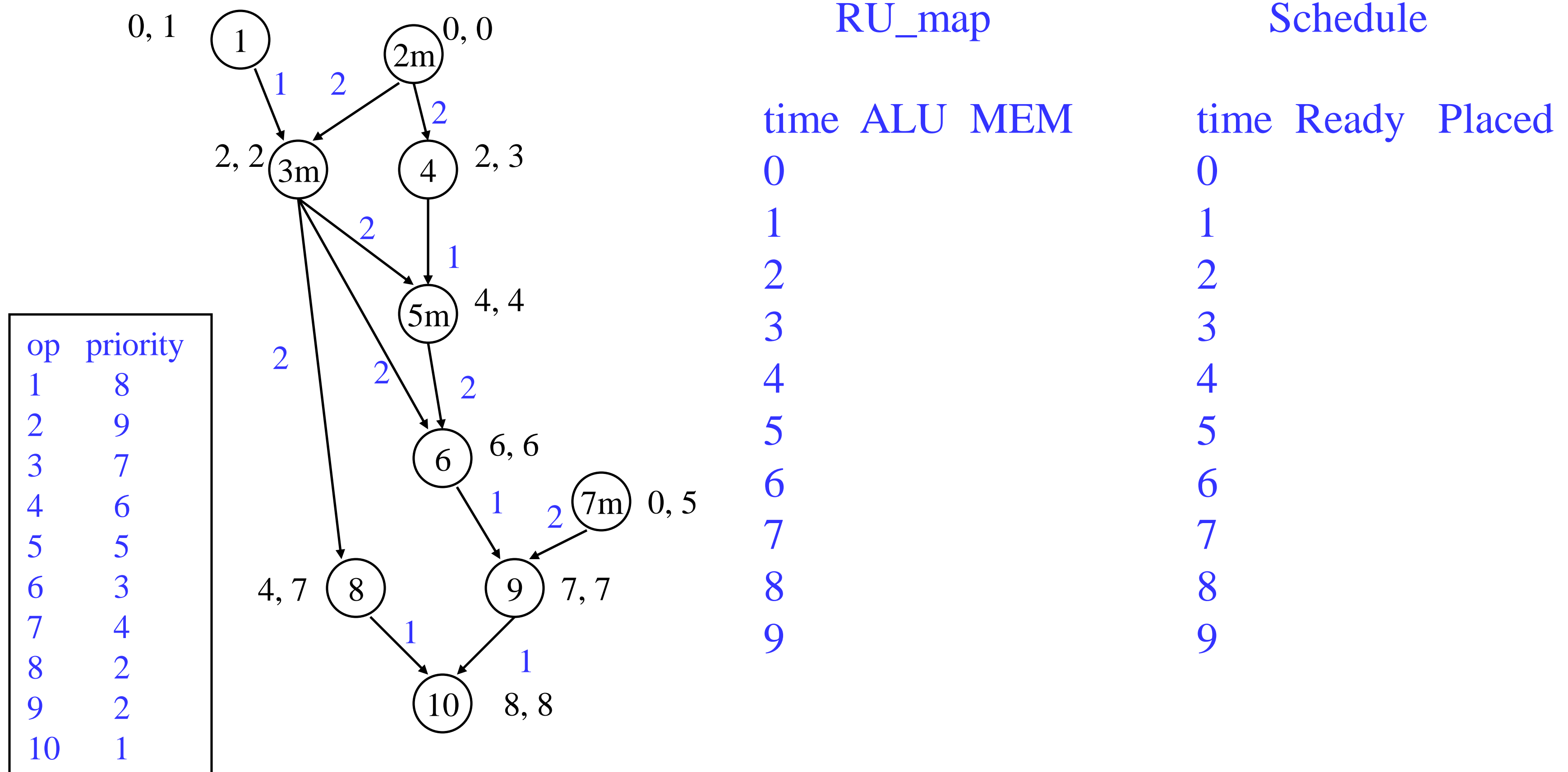


op	priority
1	8
2	9
3	7
4	6
5	5
6	3
7	4
8	2
9	2
10	1

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



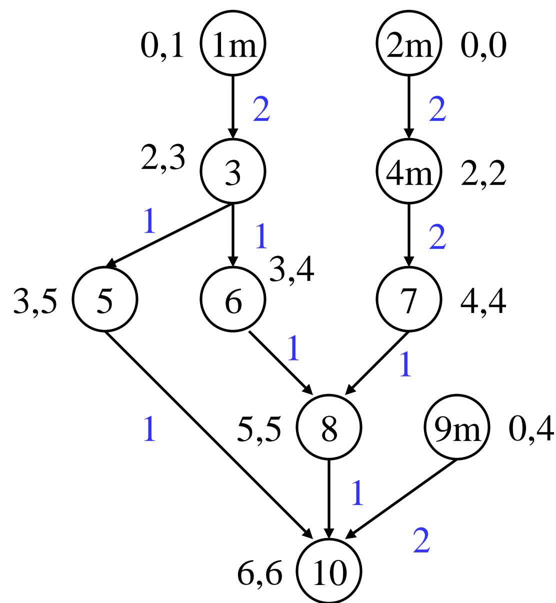
op	priority
1	8
2	9
3	7
4	6
5	5
6	3
7	4
8	2
9	2
10	1

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		

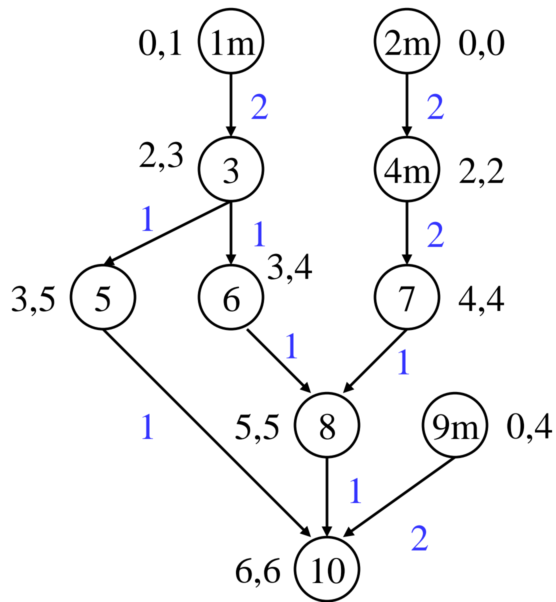
Schedule

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

1. Calculate height-based priorities
2. Schedule using Operation scheduler

Homework Problem – Answer

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



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

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

RU_map

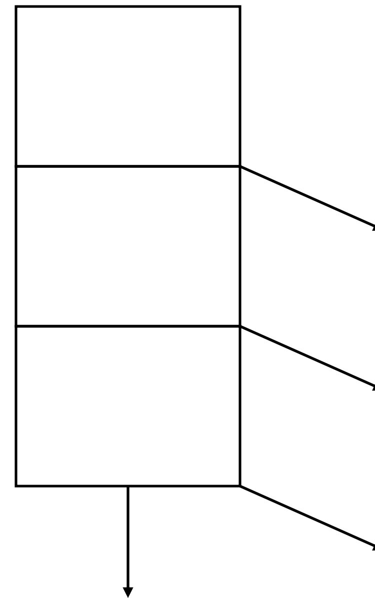
time	ALU	MEM
0		X
1		X
2		X
3	X	X
4	X	
5	X	
6	X	
7	X	
8	X	

Schedule

Time	Placed
0	2
1	1
2	4
3	3, 9
4	6
5	7
6	5
7	8
8	10

Generalize Beyond a Basic Block

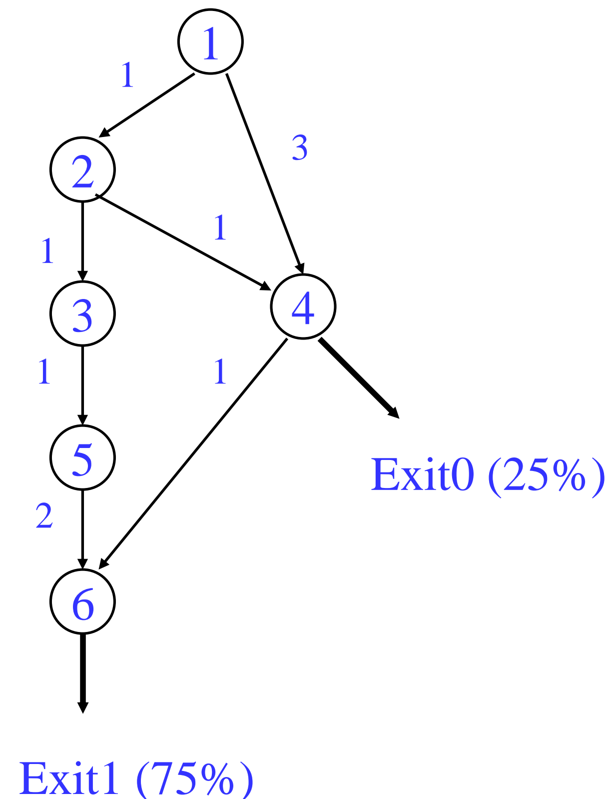
- Superblock
  - Single entry
  - Multiple exits (side exits)
  - No side entries
- Schedule just like a BB
  - Priority calculations needs change
  - Dealing with control deps



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			



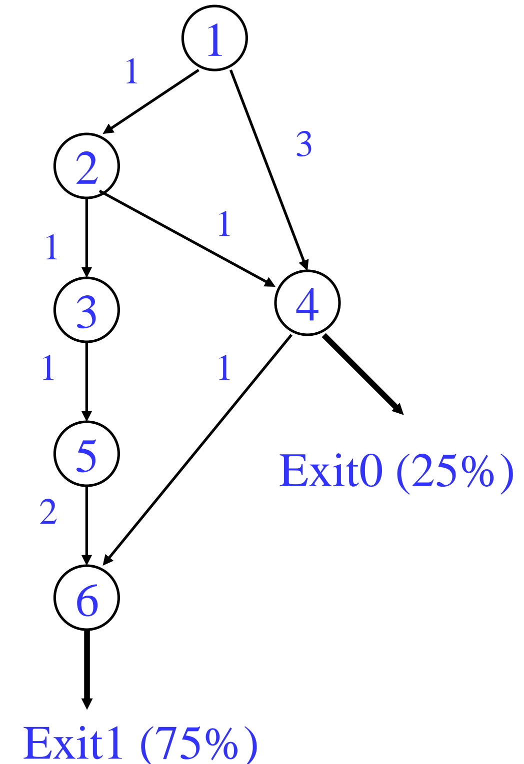
Operation Priority in a Superblock

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

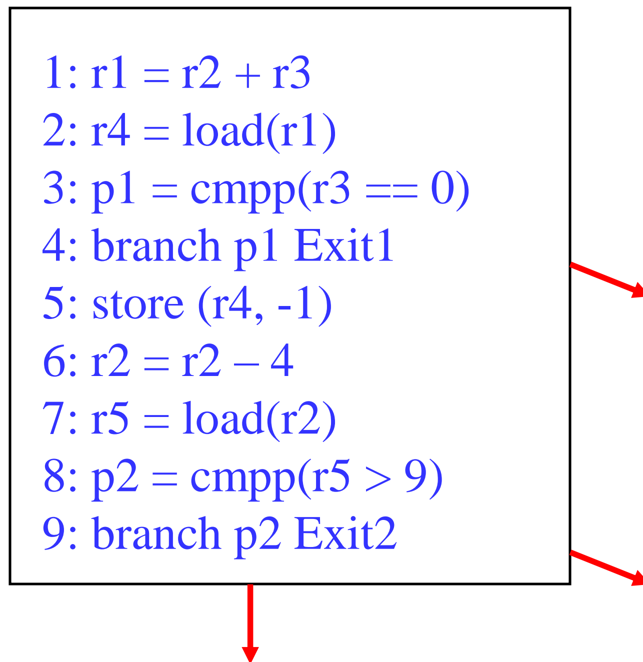
Priority(op) = SUM(Probi * (MAX_Lstart – Lstarti(op) + 1))

valid late times for op

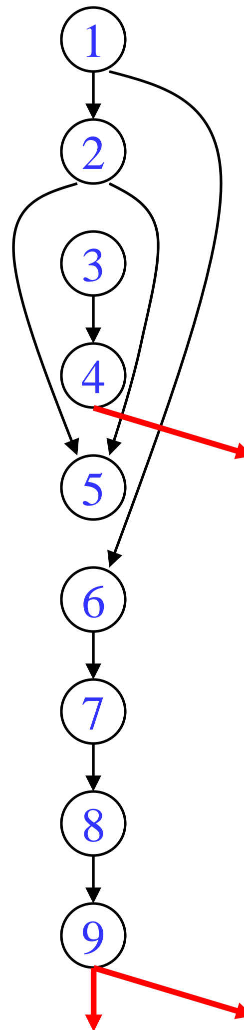
op	Lstart0	Lstart1	Priority
1			
2			
3			
4			
5			
6			



Dependences in a Superblock



Note: Control flow in red bold



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

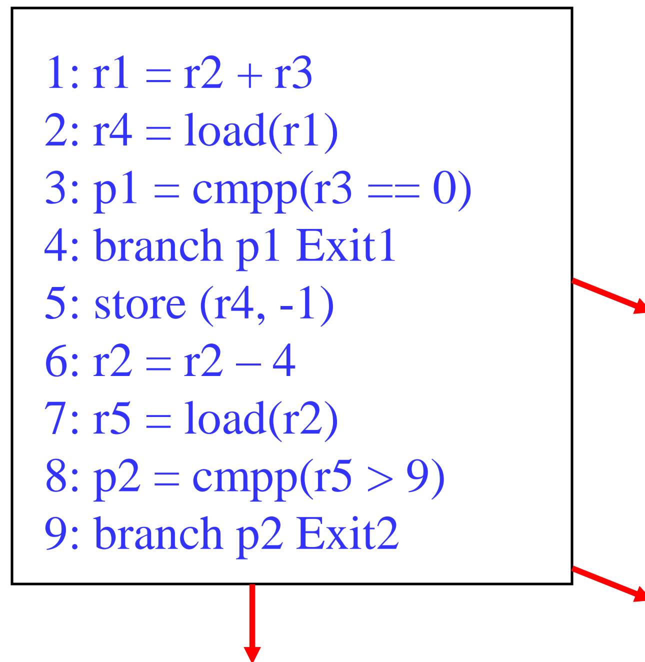
* Dependences define precedence ordering of operations to ensure correct execution semantics

* What about control dependences?

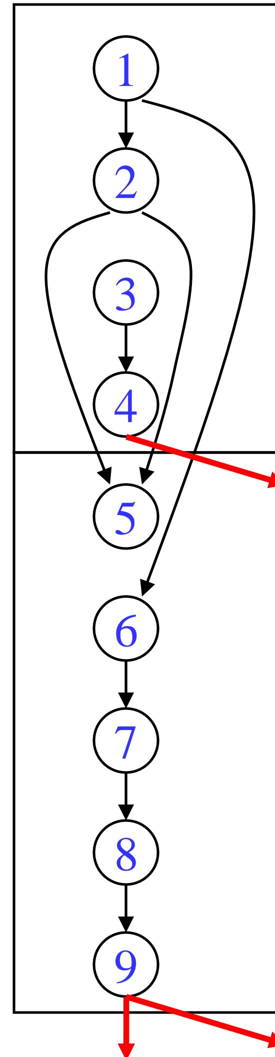
* Control dependences define precedence of ops with respect to branches

Conservative Approach to Control Dependences

Superblock



Note: Control flow in red bold



* Make branches barriers, nothing moves above or below branches

* Schedule each BB in SB separately

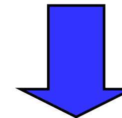
* Sequential schedules

* Whole purpose of a superblock is lost

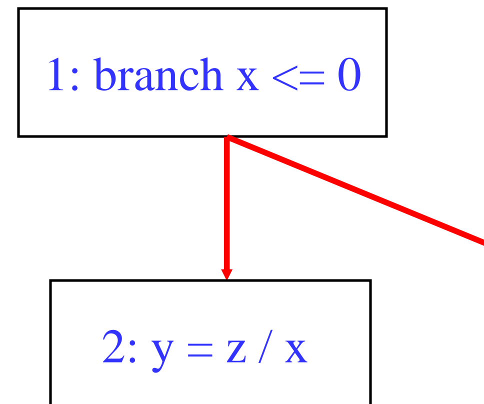
Upward Code Motion Across Branches

- 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
…
```



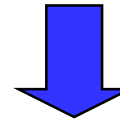
control flow graph



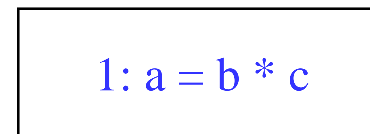
Downward Code Motion Across Branches

- 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!

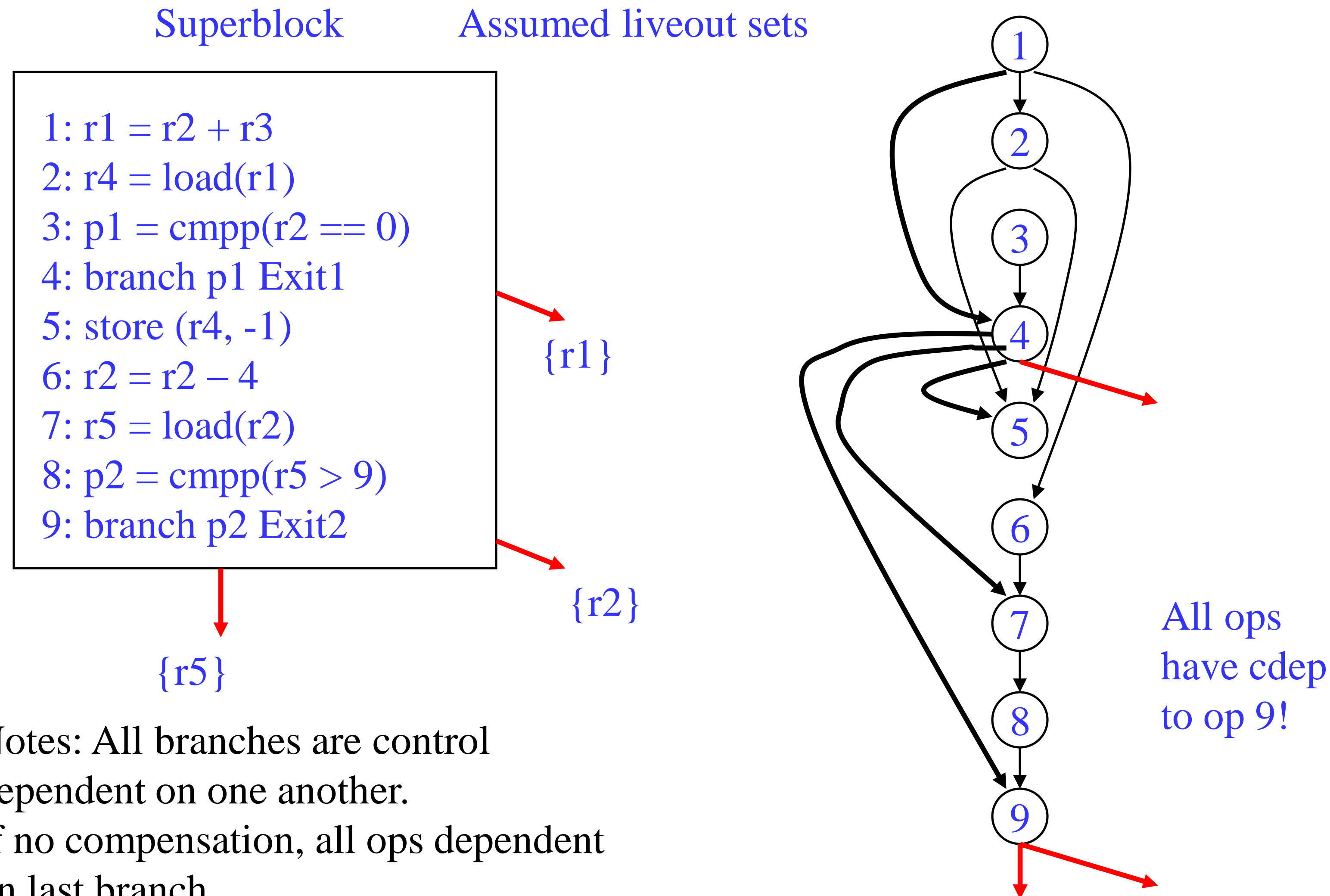
```
…
a = b * c
if (x > 0)

else
…
```



control flow graph

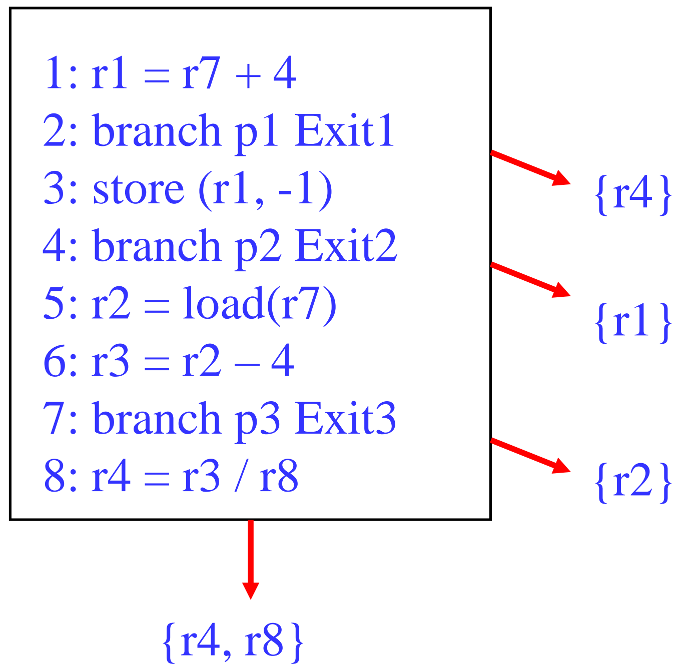


Add Control Dependences to a Superblock



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

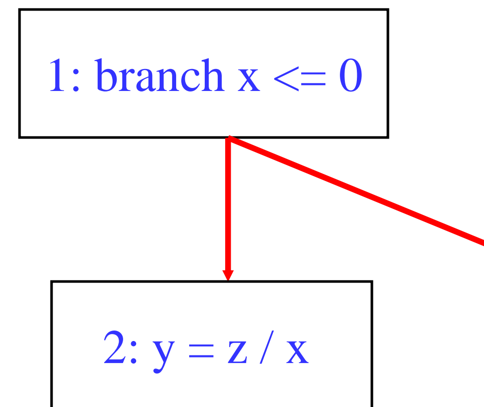
Class Problem



Draw the dependence graph

Relaxing Code Motion Restrictions

- Upward code motion is generally more effective
  - Speculate that an op is useful (just like an out-of-order processor with branch pred)
  - Start ops early, hide latency, overlap execution, more parallelism
- Removing restriction 1
  - For register ops – use register renaming
  - Could rename memory too, but generally not worth it
- Removing restriction 2
  - Need hardware support (aka <u>speculation models</u>)
    - Some ops don’t cause exceptions
    - Ignore exceptions
    - Delay exceptions



R1: y is not in liveout(1)

R2: op 2 will never cause an exception when op1 is taken

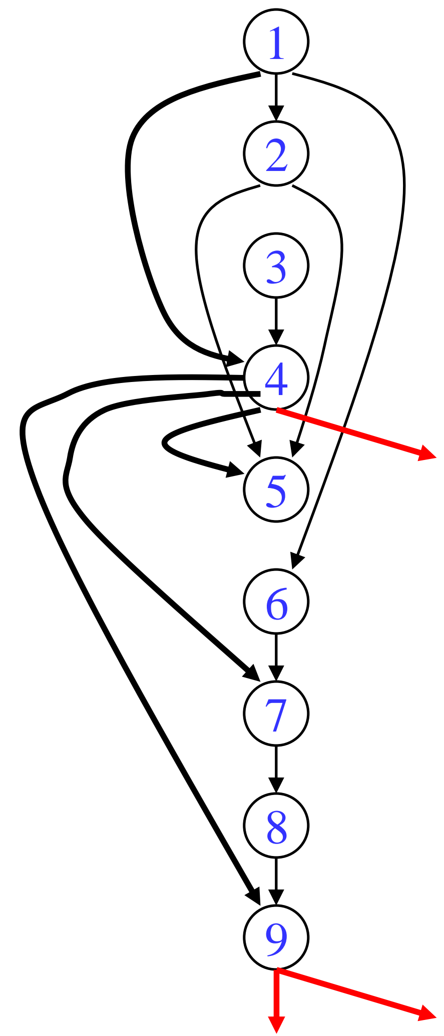
Restricted Speculation Model

- ❖ Most processors have 2 classes of opcodes
  - » Potentially exception causing
    - • load, store, integer divide, floating-point
  - » Never excepting
    - • Integer add, multiply, etc.
    - • Overflow is detected, but does not terminate program execution
- ❖ Restricted model
  - » R2 only applies to potentially exception causing operations
  - » Can freely speculate all never exception ops (still limited by R1 however)

```
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
```

{r1}

{r2}

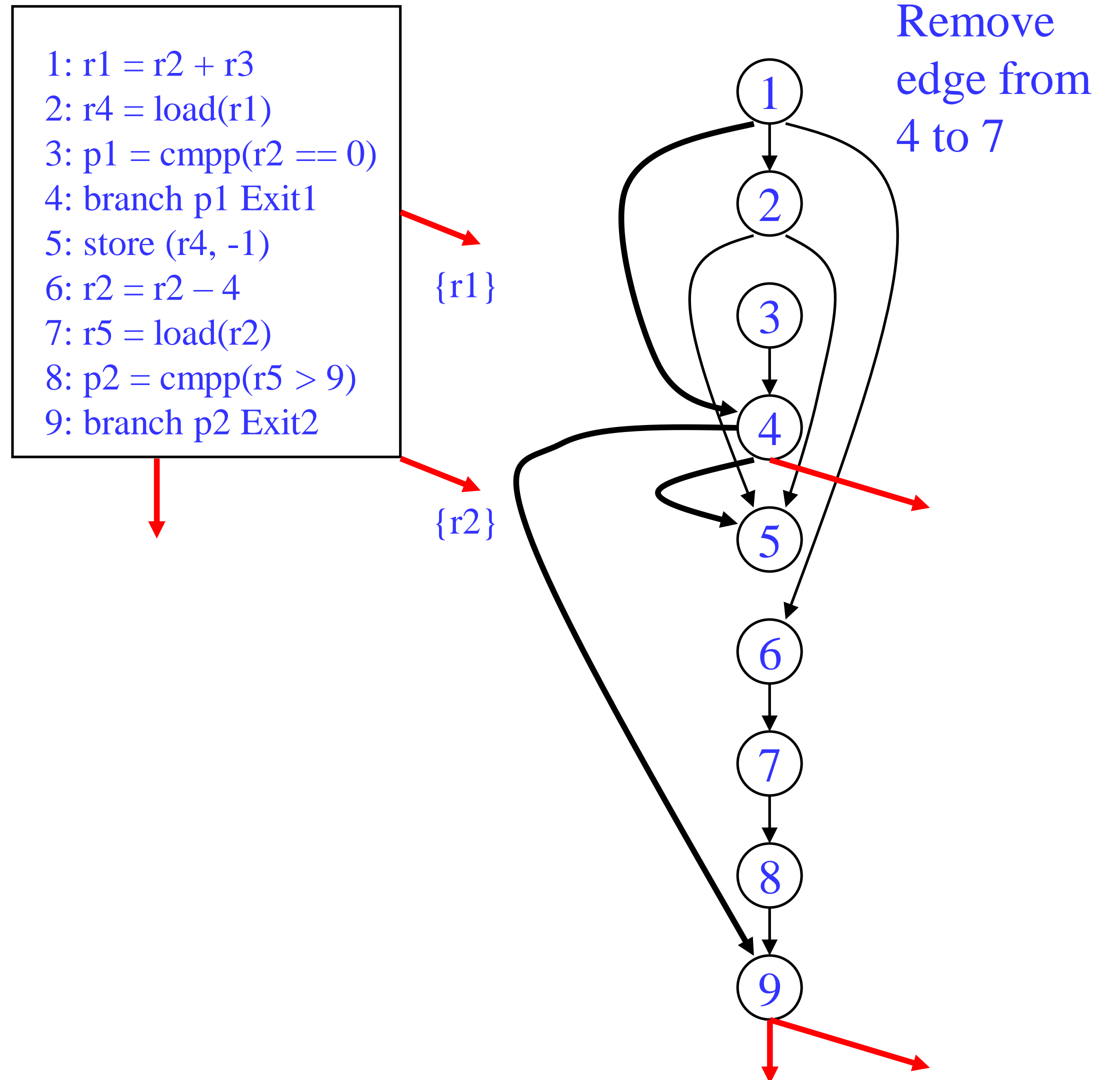


We assumed restricted speculation when this graph was drawn.

This is why there is no cdep between 4 → 6 and 4→ 8

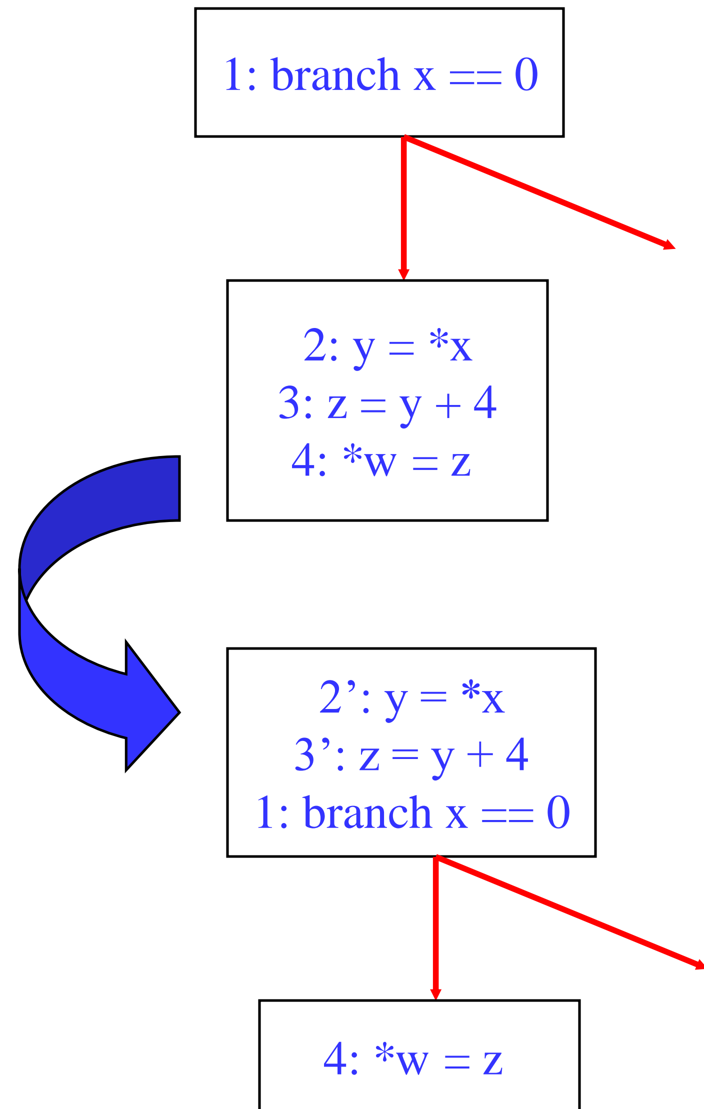
General Speculation Model

- 2 types of exceptions
  - Program terminating (traps)
    - Div by 0, illegal address
  - Fixable (normal and handled at run time)
    - Page fault, TLB miss
- General speculation
  - Processor provides non-trapping versions of all operations (div, load, etc)
  - Return some bogus value (0) when error occurs
  - R2 is completely ignored, only R1 limits speculation
  - Speculative ops converted into non-trapping version
  - Fixable exceptions handled as usual for non-trapping ops

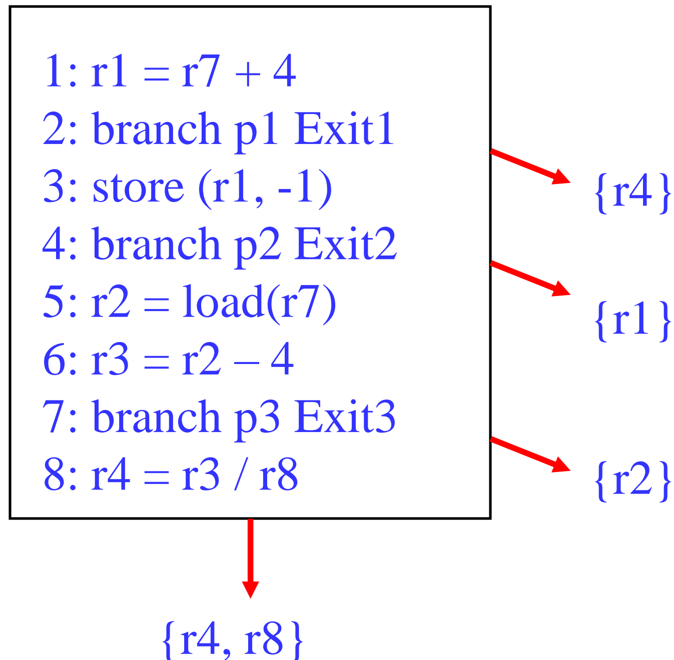


Programming Implications of General Spec

- Correct program
  - No problem at all
  - Exceptions will only result when branch is taken
  - Results of excepting speculative operation(s) will not be used for anything useful (R1 guarantees this!)
- Program debugging
  - Non-trapping ops make this almost impossible
  - Disable general speculation during program debug phase



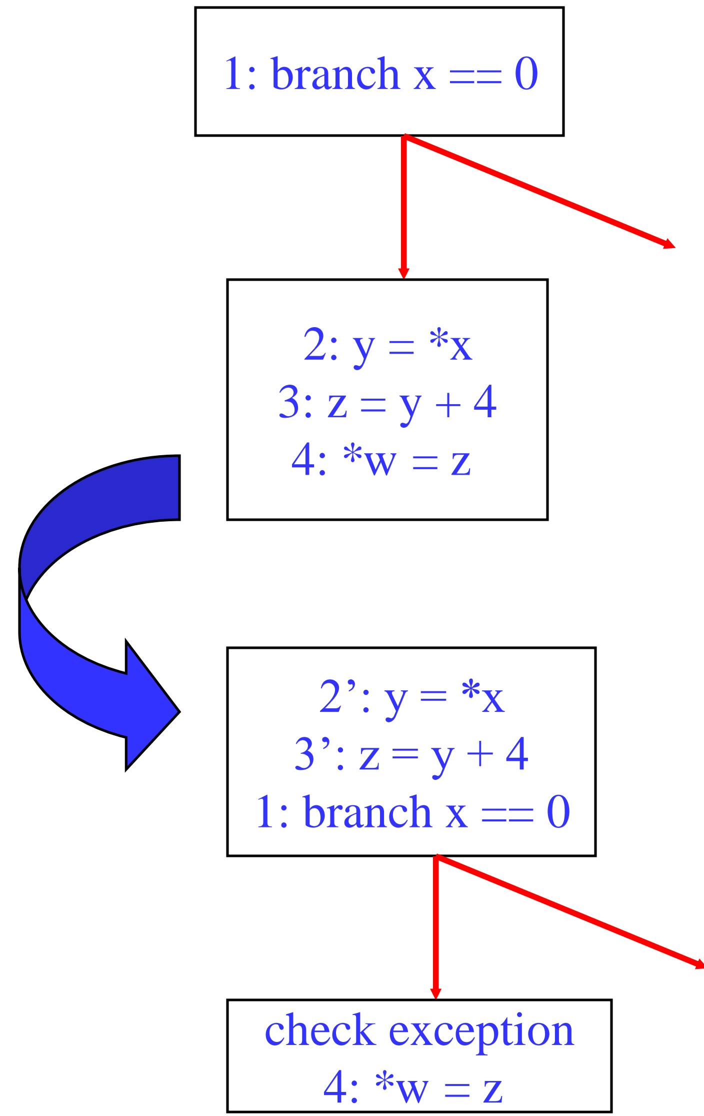
Class Problem



1. Starting with the graph assuming restricted speculation, what edges can be removed if general speculation support is provided?
2. With more renaming, what dependences could be removed?

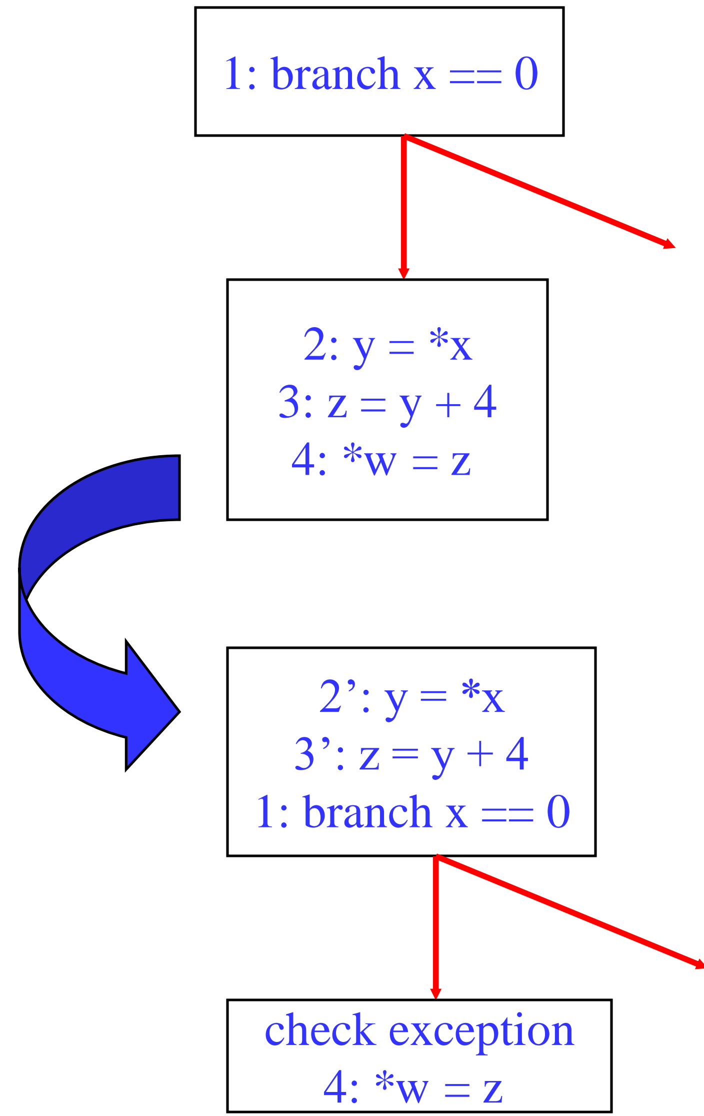
Sentinel Speculation Model

- Ignoring all speculative exceptions is painful
  - » Debugging issue (is a program ever fully correct?)
- Also, handling of all fixable exceptions for speculative ops can be slow
  - » Extra page faults
- Sentinel speculation
  - » Mark speculative ops (opcode bit)
  - » Exceptions for speculative ops are noted, but not handed immediately (return garbage value)
  - » Check for exception conditions in the "home block" of speculative potentially excepting ops



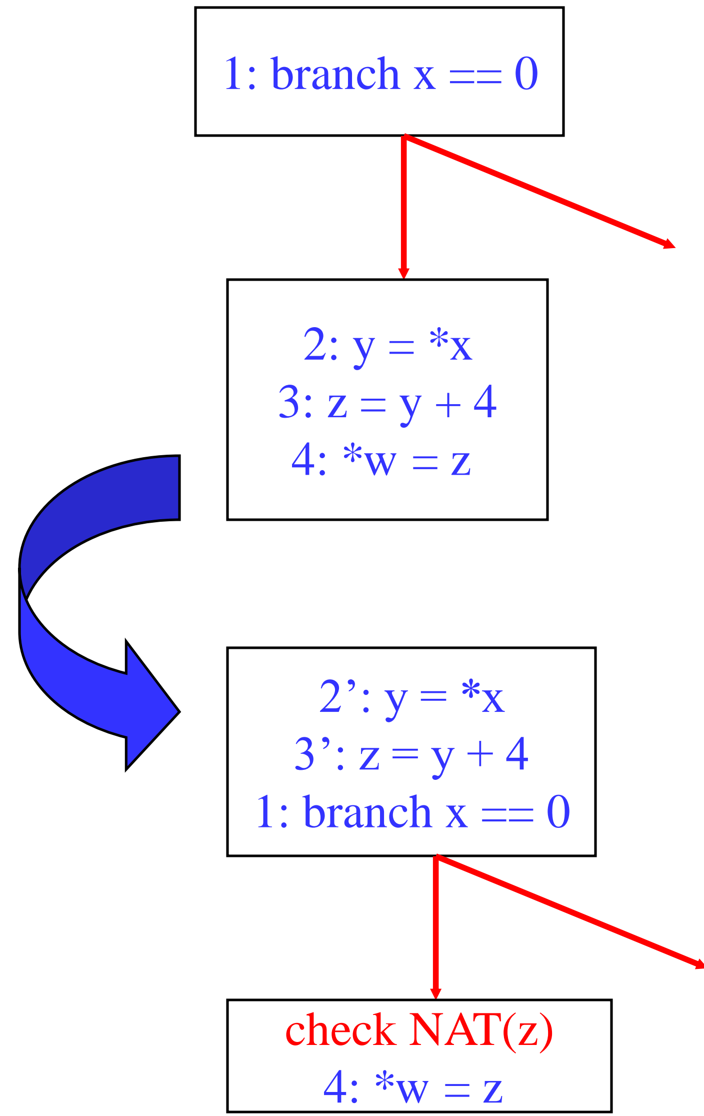
Delaying Speculative Exceptions

- 3 things needed
  - Record exceptions
  - Check for exceptions
  - Regenerate exception
    - Re-execute ops including dependent ops
    - Terminate execution or process exception
- Recording them
  - Extend every register with an extra bit
    - Exception tag (or NAT bit)
    - Reg data is garbage when set
    - Bit is set when either
      - Speculative op causes exception
      - Speculative op has a NAT'd source operand (exception propagation)



Delaying Speculative Exceptions (2)

- Check for exceptions
  - Test NAT bit of appropriate register (last register in dependence chain) in home block
  - Explicit checks
    - Insert new operation to check NAT
  - Implicit checks
    - Non-speculative use of register automatically serves as NAT check
- Regenerate exception
  - Figure out the exact cause
  - Handle if possible
  - Check with NAT condition branches to “recovery code”
  - Compiler generates the recovery code specific to each check

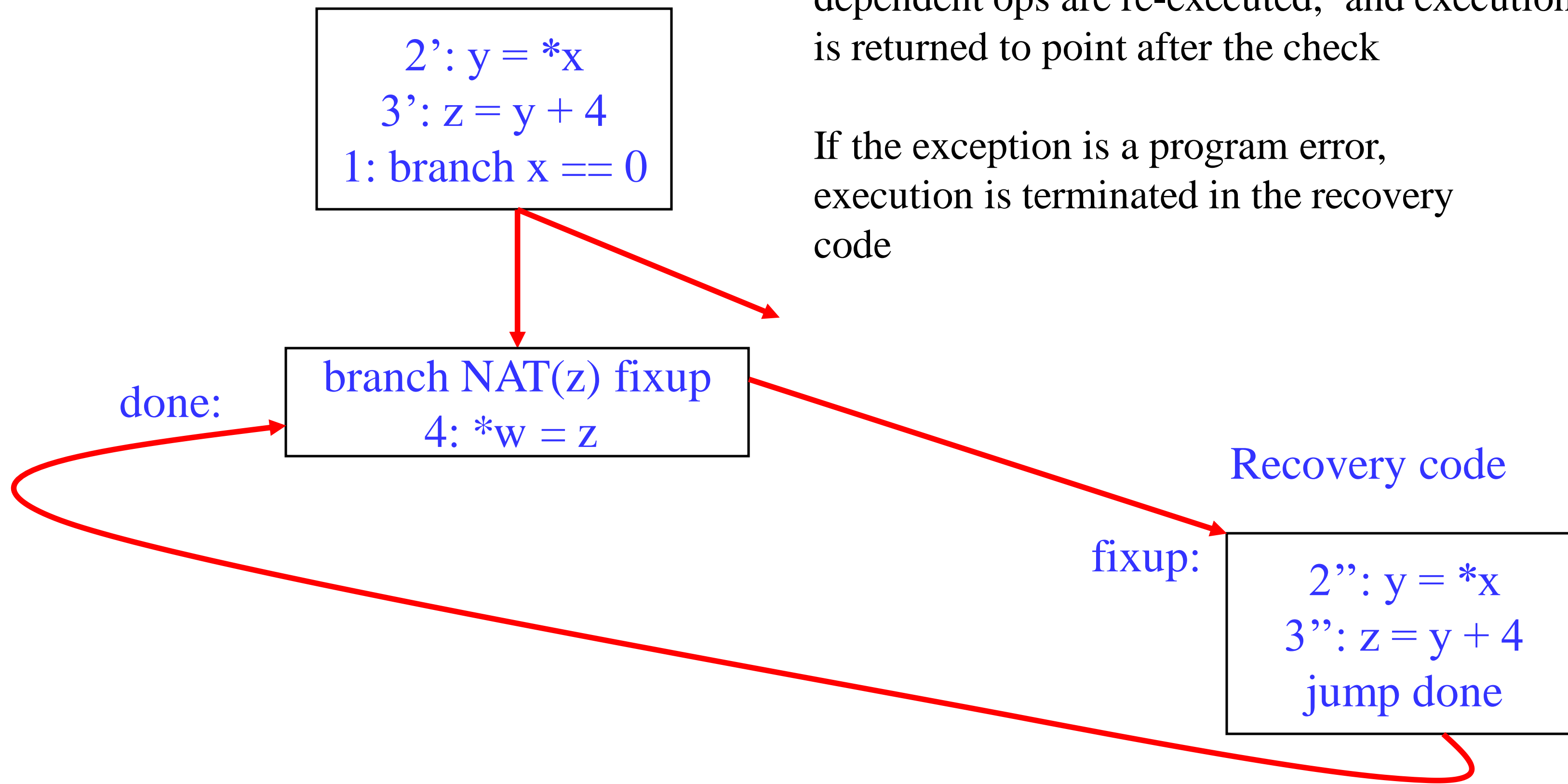


Delaying Speculative Exceptions (3)

In recovery code, the exception condition will be regenerated as the excepting op is re-executed with the same inputs

If the exception can be handled, it is, all dependent ops are re-executed, and execution is returned to point after the check

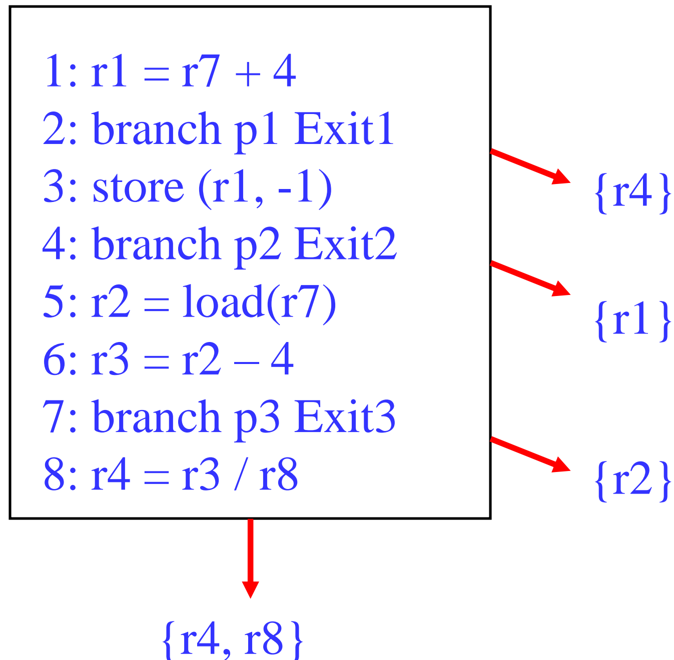
If the exception is a program error, execution is terminated in the recovery code



Implicit vs Explicit Checks

- Explicit
  - Essentially just a conditional branch
  - Nothing special needs to be added to the processor
  - Problems
    - Code size
    - Checks take valuable resources
- Implicit
  - Use existing instructions as checks
  - Removes problems of explicit checks
  - However, how do you specify the address of the recovery block?, how is control transferred there?
  - Hardware table
    - Indexed by PC
    - Indicates where to go when NAT is set
- IA-64 uses explicit checks

Homework Problem



1. Move ops 5, 6, 8 as far up in the SB as possible assuming sentinel speculation support and register renaming
2. Insert the necessary checks and recovery code (assume ld, st, and div can cause exceptions)