EECS 583 – Class 12
Superblock Scheduling
Software Pipelining Intro

University of Michigan

October 17, 2018

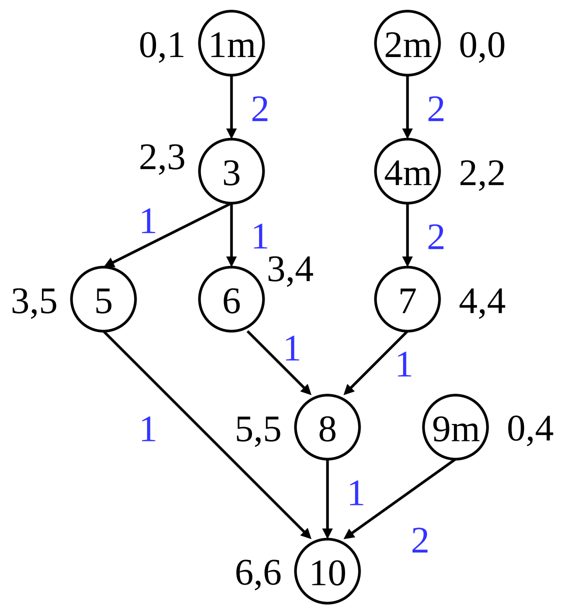
Announcements + Reading Material

- Project discussion meetings
  - No class next week (Oct 22 & 24)
  - Each group meets 15 mins with Ze and I
  - Signup today in class, signup sheet on my door (4633 BBB) if you miss class or can't decide on a timeslot
  - Be prompt, show up a few minutes early as back-to-back meetings
- Project proposals
  - Due Wednesday, Oct 31, 11:59pm
  - 1 paragraph summary of what you plan to work on
    - Topic, approach, objective
    - 1-2 references
  - Email to me and Ze, cc your group members
- Today's class reading
  - "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops", B. Rau, MICRO-27, 1994, pp. 63-74.
- Next next Monday's reading
  - "Code Generation Schema for Modulo Scheduled Loops", B. Rau, M. Schlansker, and P. Tirumalai, MICRO-25, Dec. 1992.

Homework Problem From Last Time – Answer

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

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	

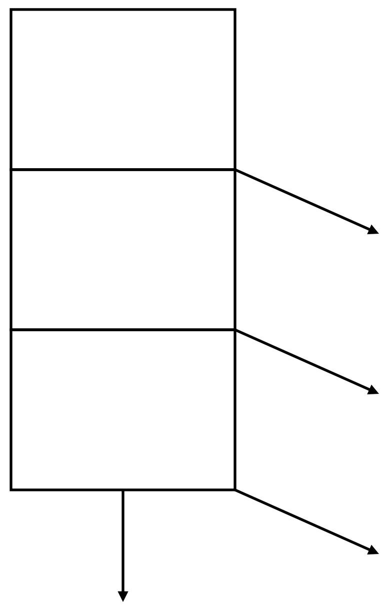
Schedule

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

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

Generalize Beyond a Basic Block

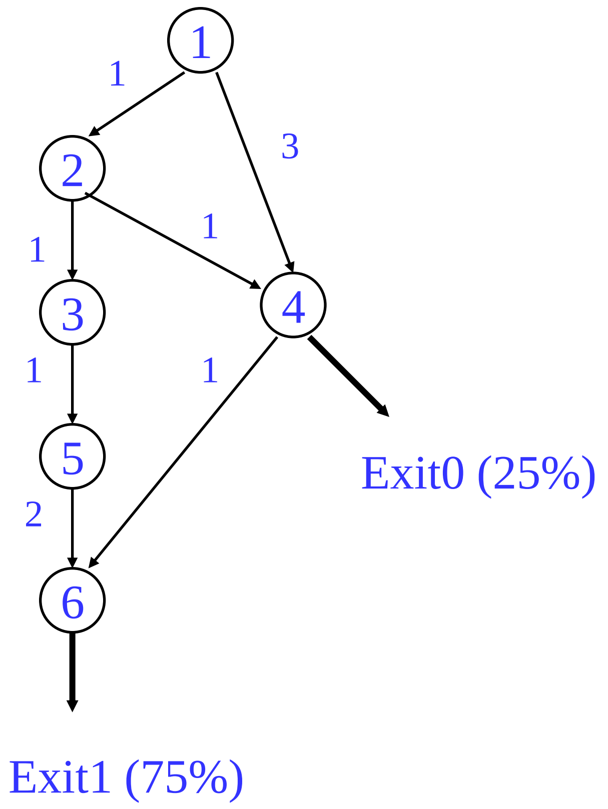
- Superblock
  - Single entry
  - Multiple exits (side exits)
  - No side entries
- Schedule just like a BB
  - Priority calculations needs change
  - Dealing with control deps



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			

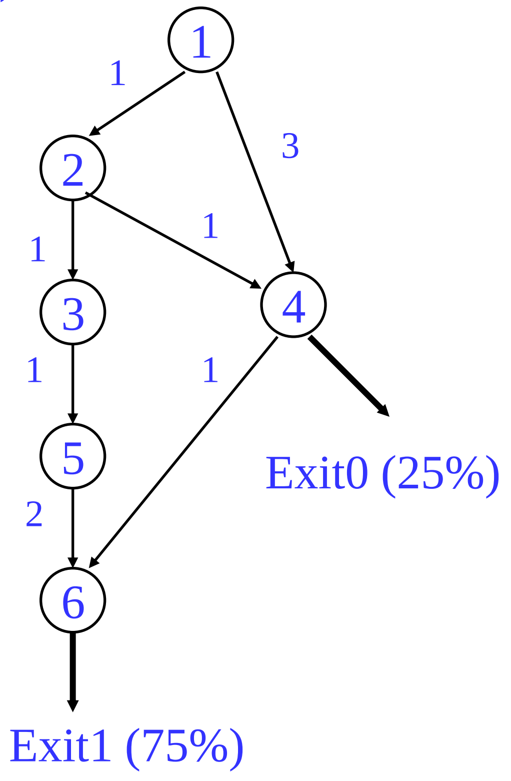


Operation Priority in a Superblock

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

$$\text{Priority}(op) = \underbrace{\text{SUM}}_{\text{valid late times for op}}(\text{Prob}_i * (\text{MAX\_Lstart} - \text{Lstart}_i(op) + 1))$$

op	Lstart0	Lstart1	Priority
1			
2			
3			
4			
5			
6			

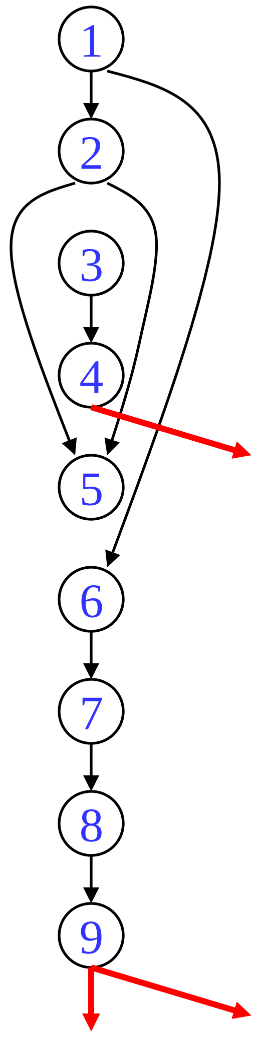


Dependences in a Superblock

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$\rightarrow$ 6 is reg anti

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

* What about control dependences?

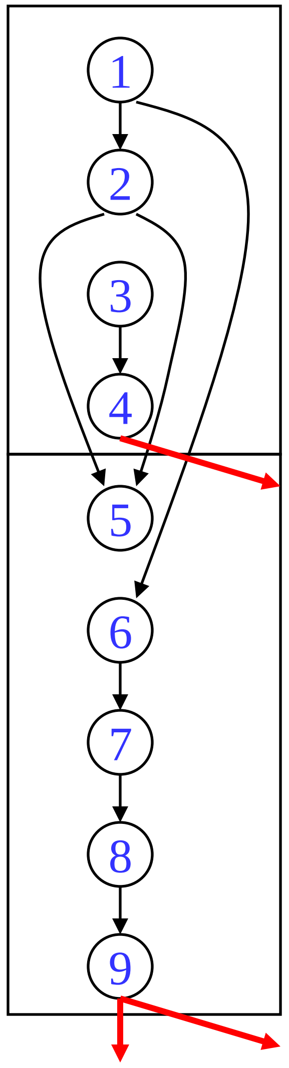
* Control dependences define precedence of ops with respect to branches

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

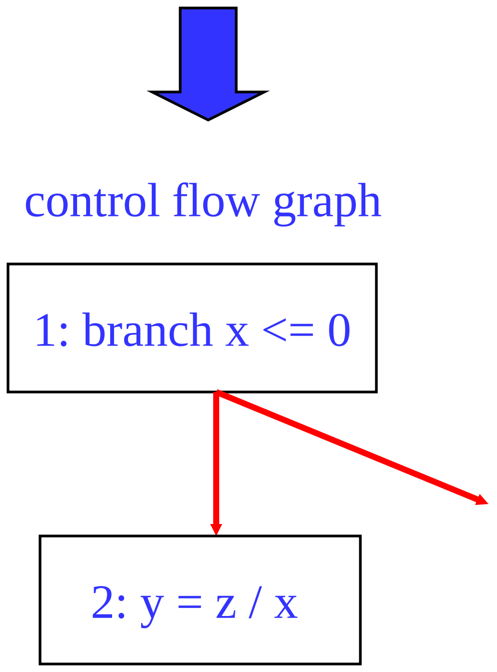
* Sequential schedules

* 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
…
```



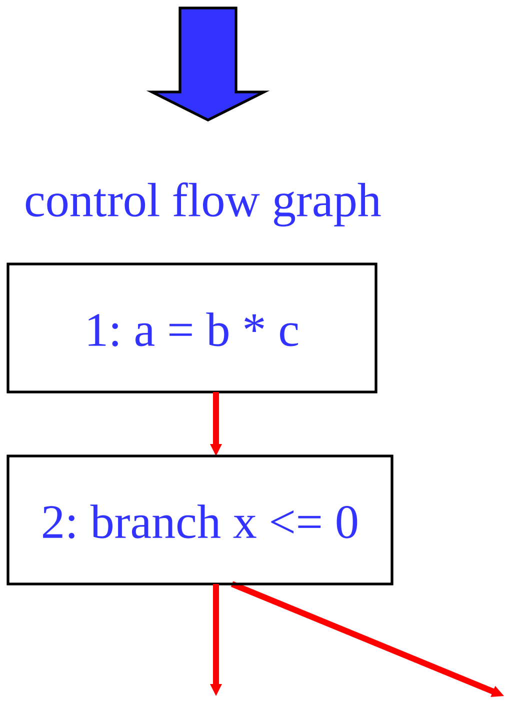
Downward Code Motion Across Branches

- 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

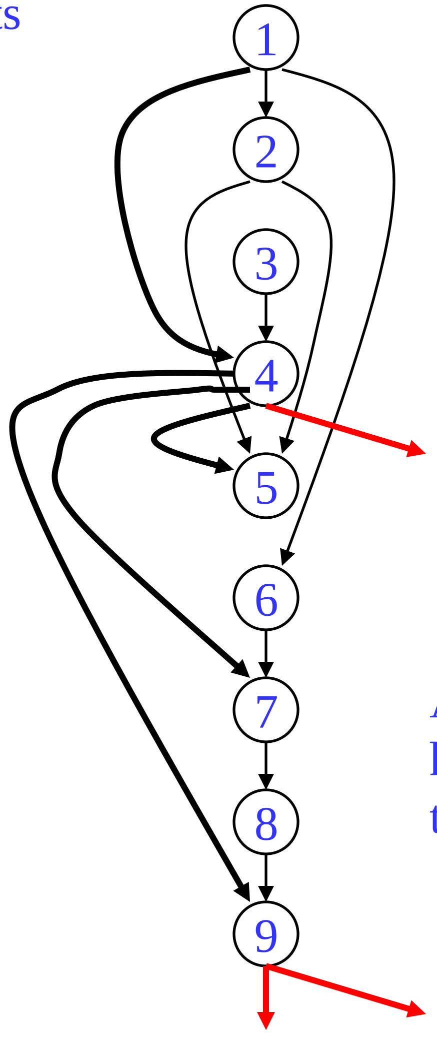
Assumed liveout sets

```
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}

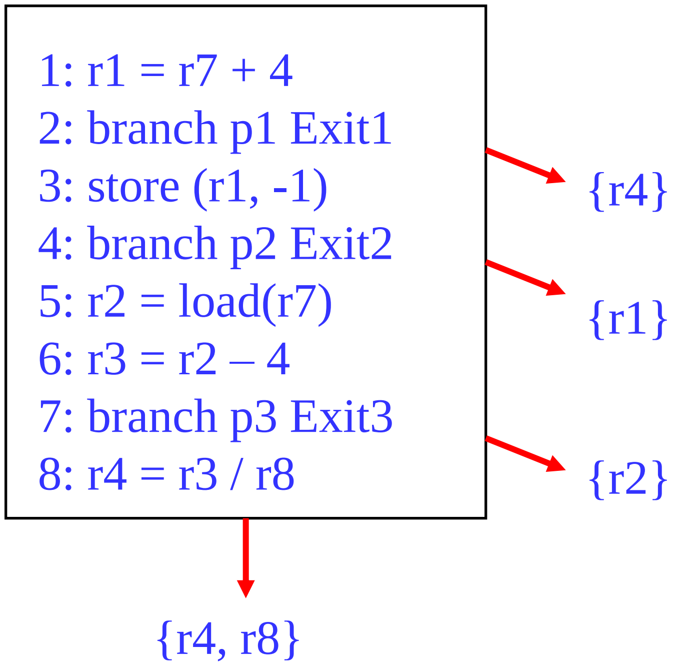
{r5}



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

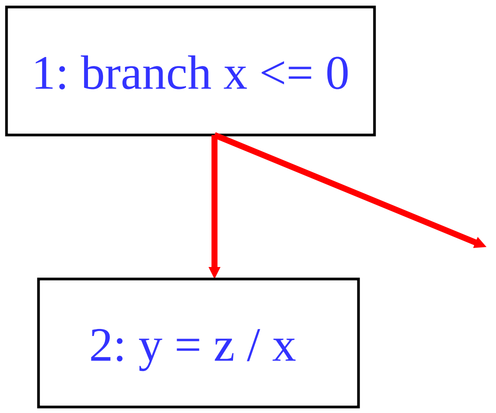


```
1: r1 = r7 + 4
2: branch p1 Exit1
3: store (r1, -1)
4: branch p2 Exit2
5: r2 = load(r7)
6: r3 = r2 – 4
7: branch p3 Exit3
8: r4 = r3 / r8
```

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



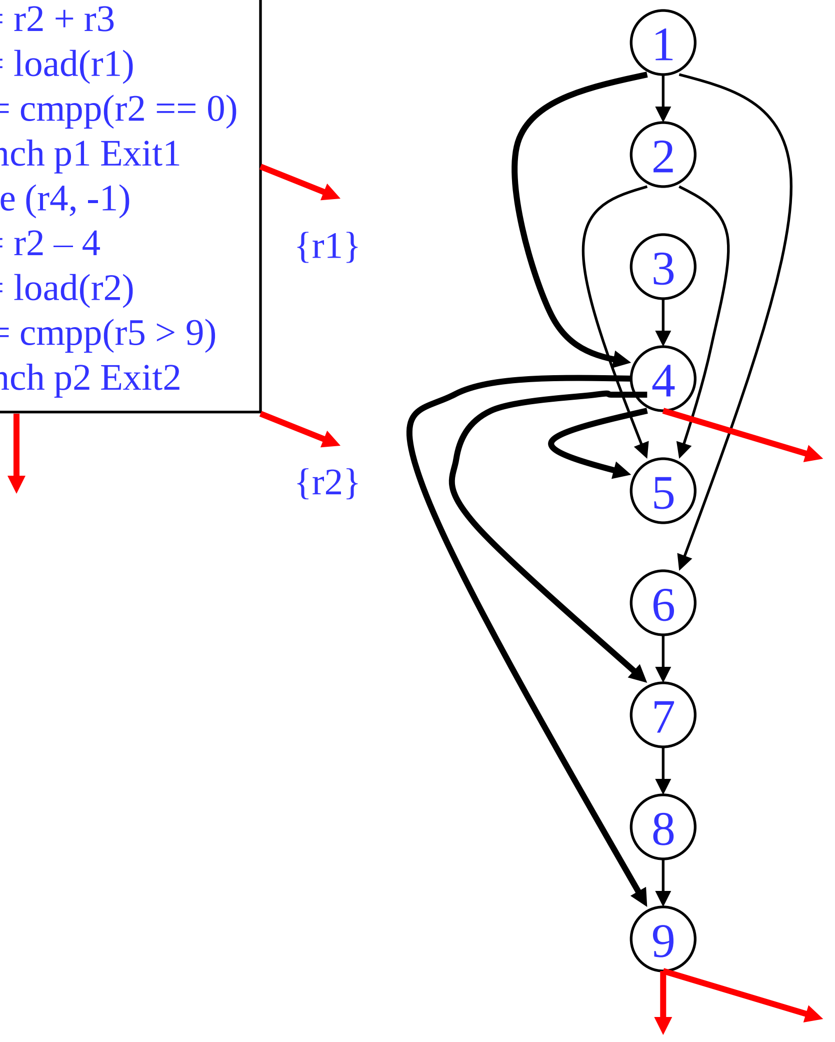
R1: y is not in liveout(1)

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
```



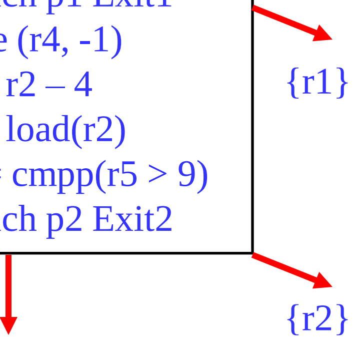
We assumed restricted speculation when this graph was drawn.

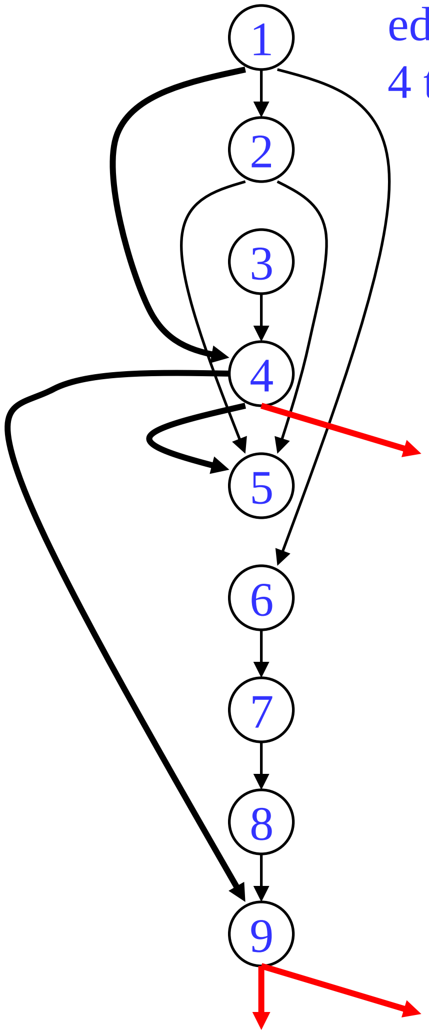
This is why there is no cdep between 4 → 6 and 4 → 8

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

```
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
```

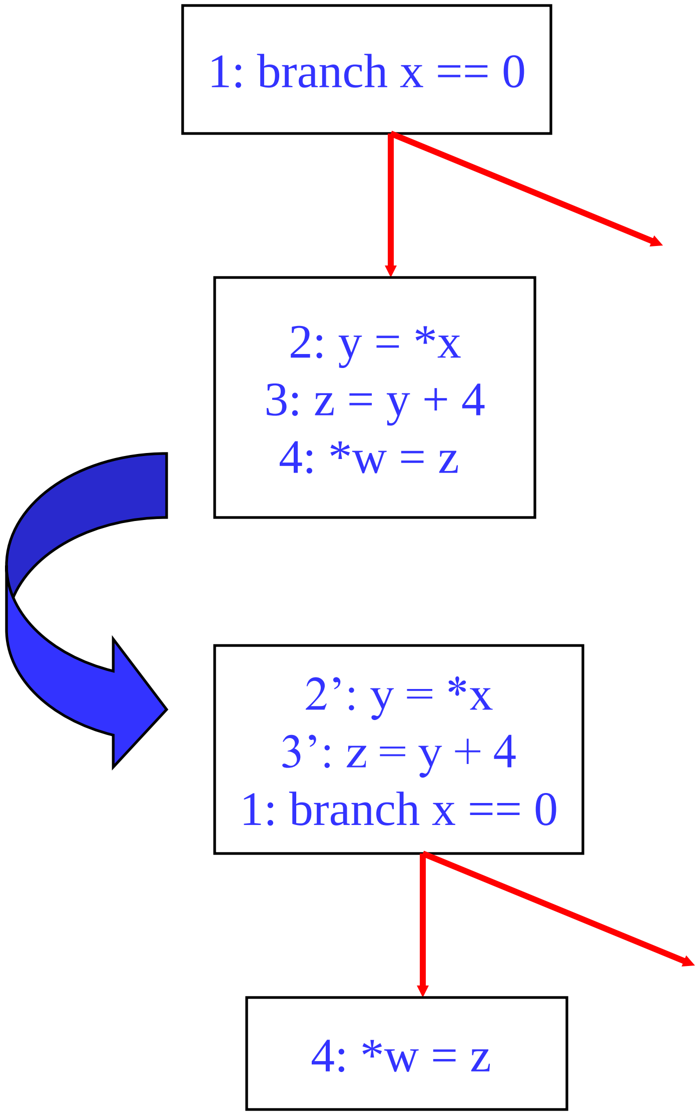




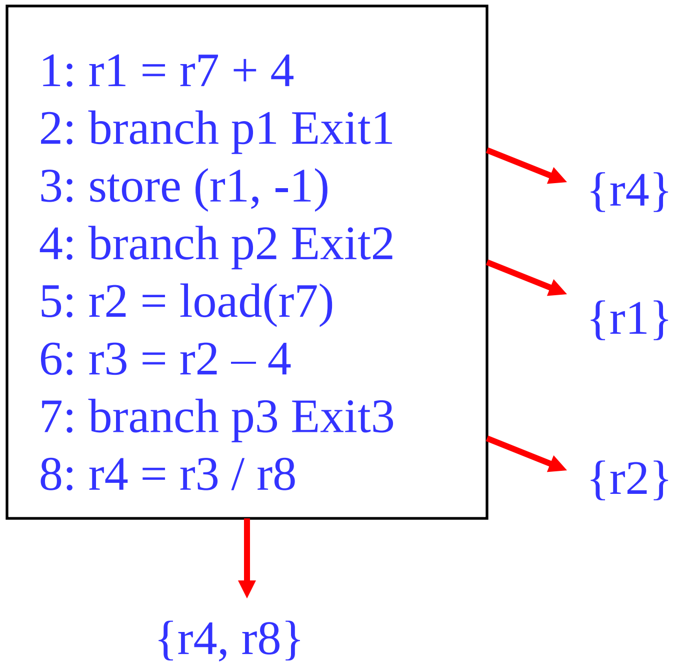
Remove edge from 4 to 7

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



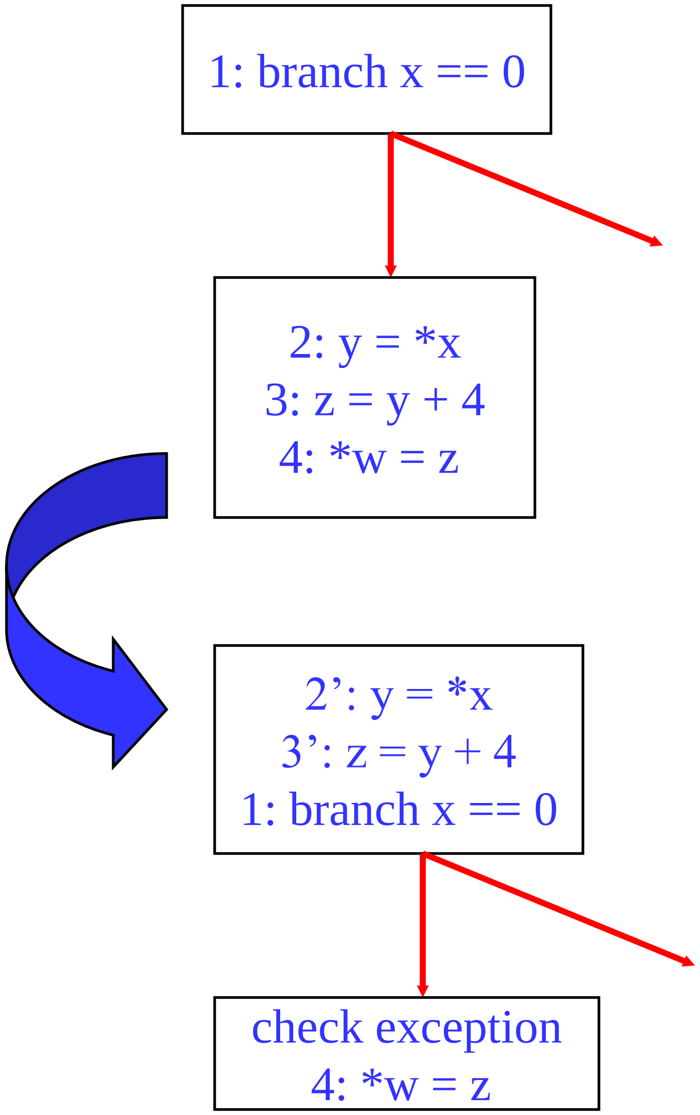
Class Problem



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?

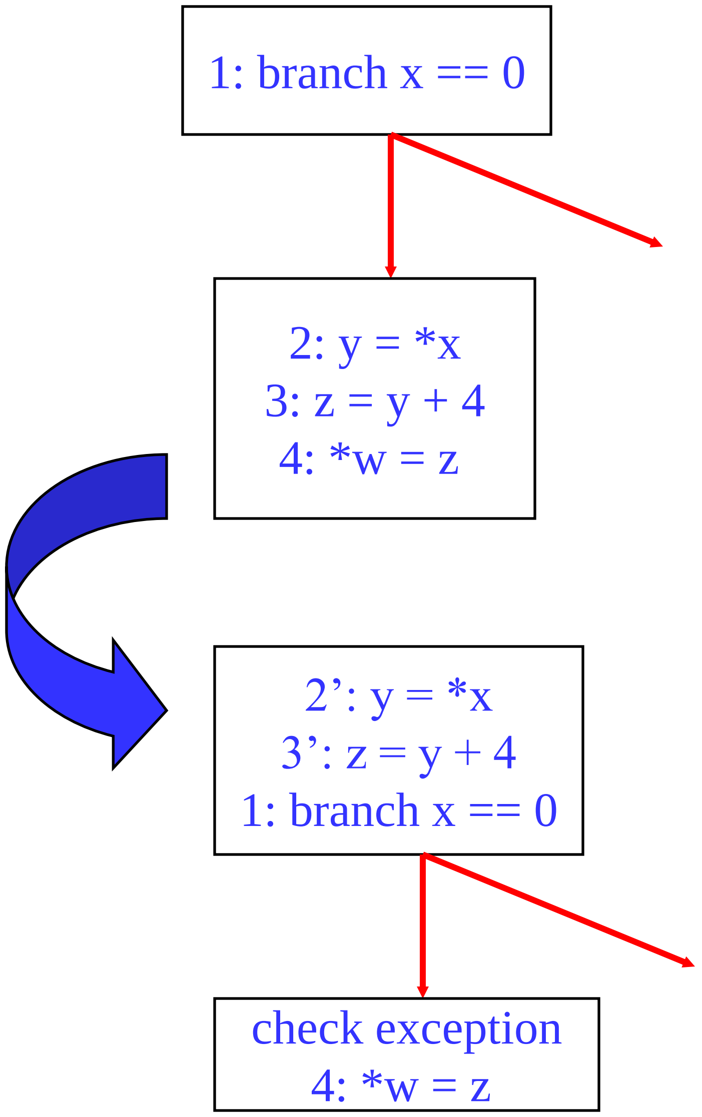
Sentinel Speculation Model

- 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



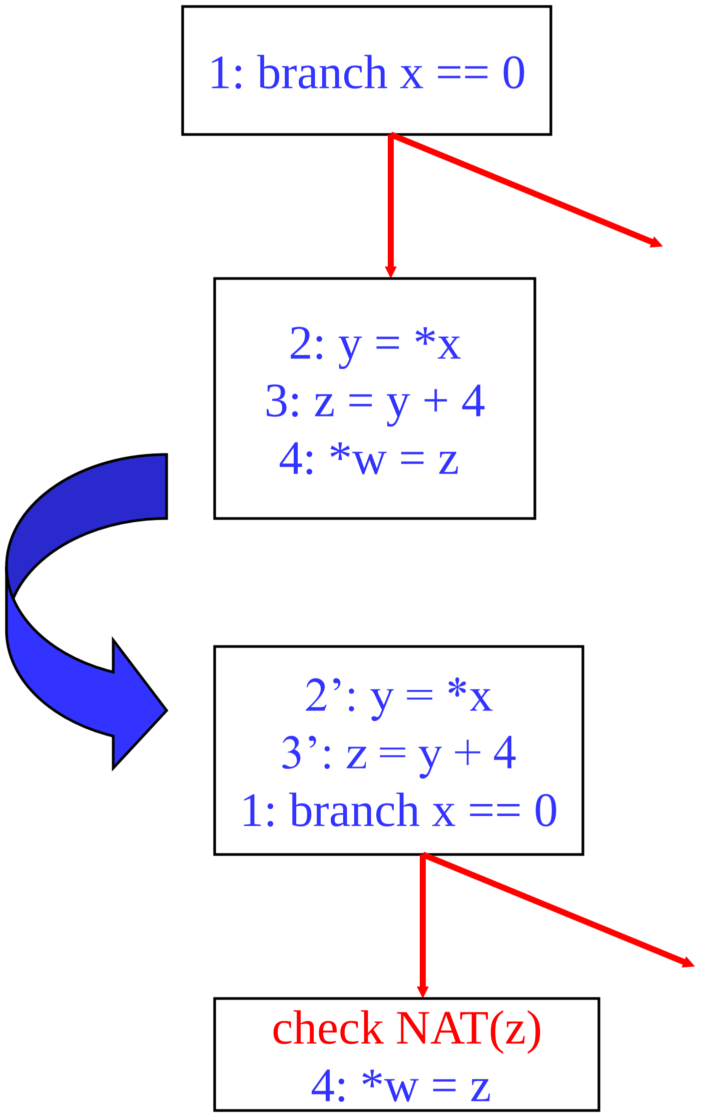
Delaying Speculative Exceptions

- 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



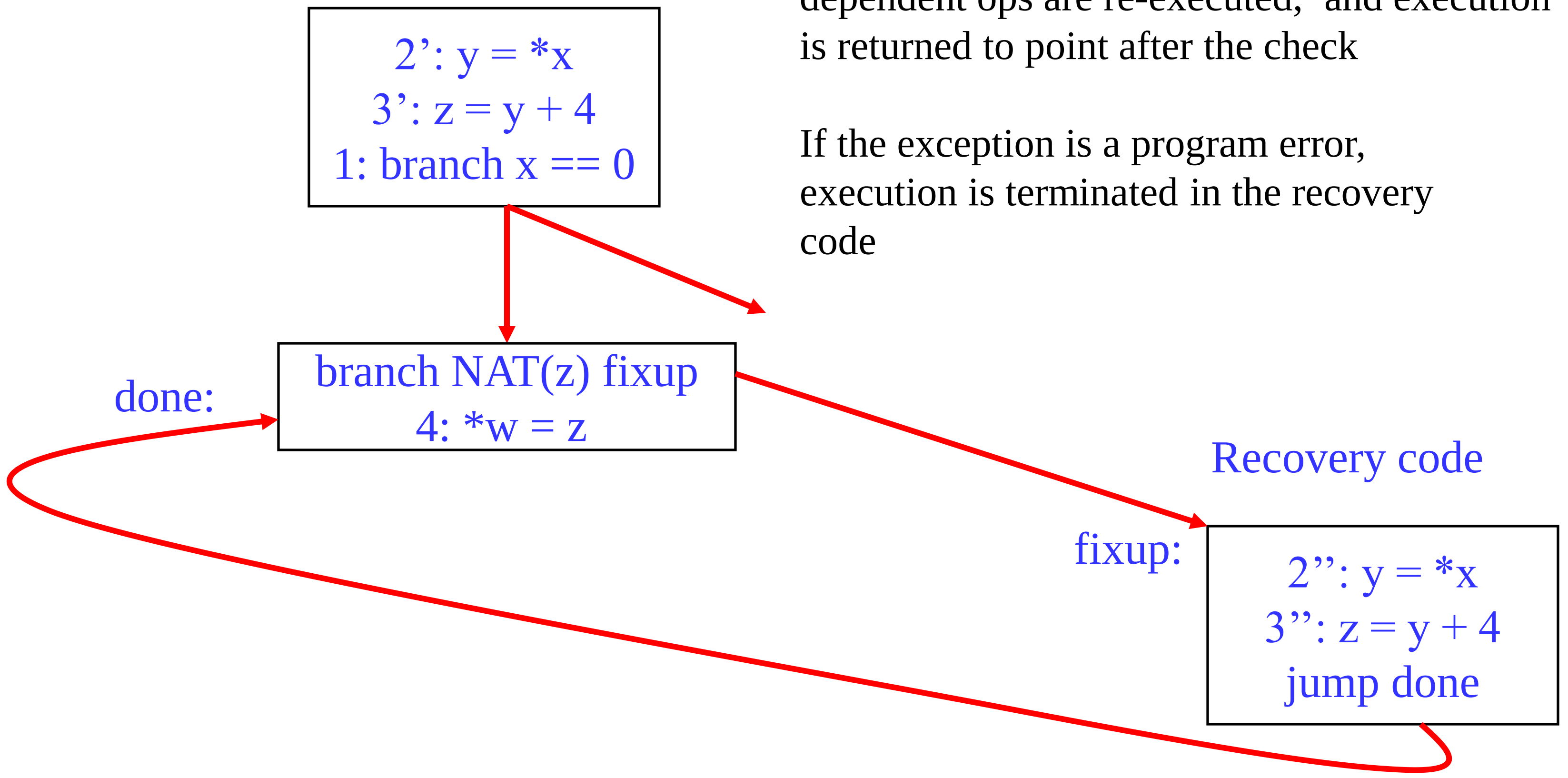
Delaying Speculative Exceptions (3)

Recovery code consists of chain of operations starting with a potentially excepting speculative op up to its corresponding check

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

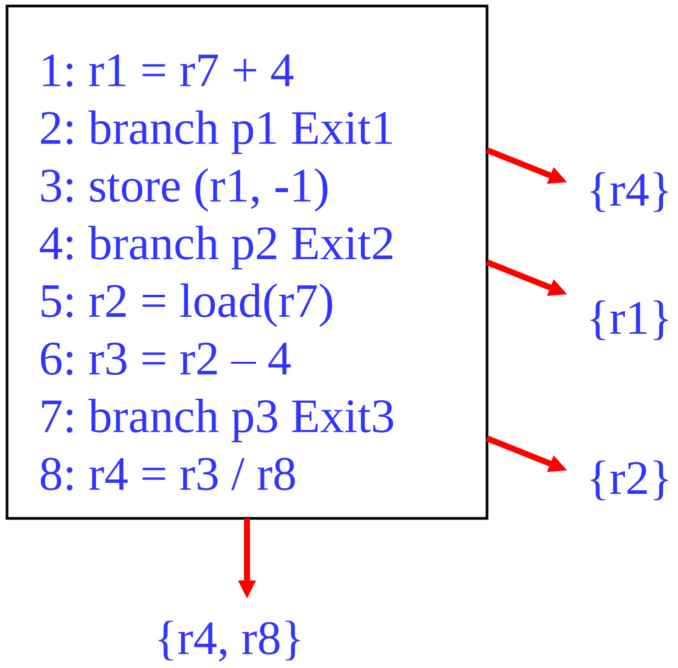
If the exception is a program error, execution is terminated in the recovery code



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: r1 = r7 + 4
2: branch p1 Exit1
3: store (r1, -1)
4: branch p2 Exit2
5: r2 = load(r7)
6: r3 = r2 – 4
7: branch p3 Exit3
8: r4 = r3 / r8
```

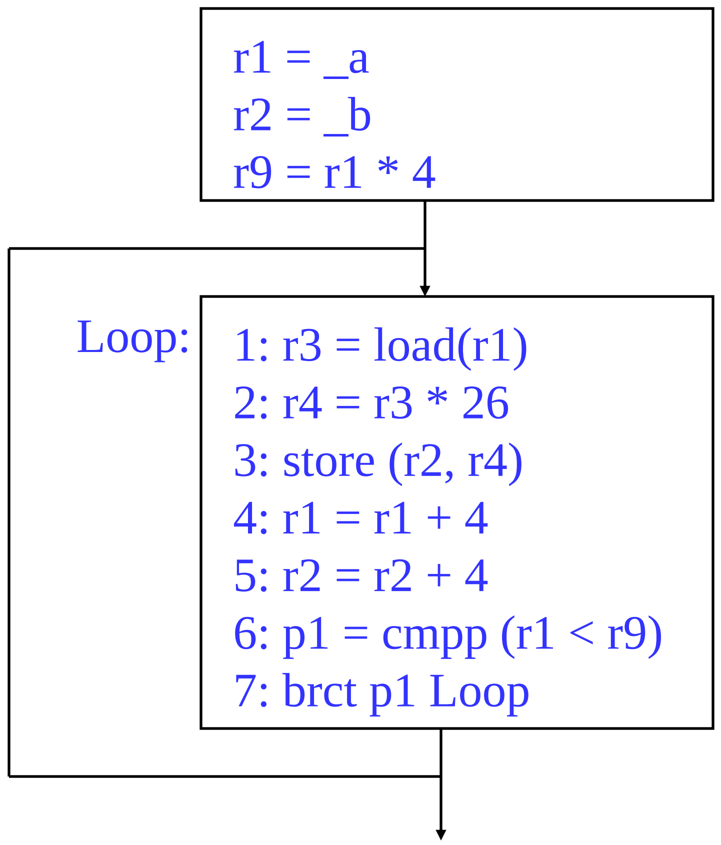
1. Move ops 5, 6, 8 as far up in the SB as possible assuming sentinel speculation support and register renaming
2. Insert the necessary checks and recovery code (assume ld, st, and div can cause exceptions)

Change Focus to Scheduling Loops

Most of program execution time is spent in loops

Problem: How do we achieve compact schedules for loops

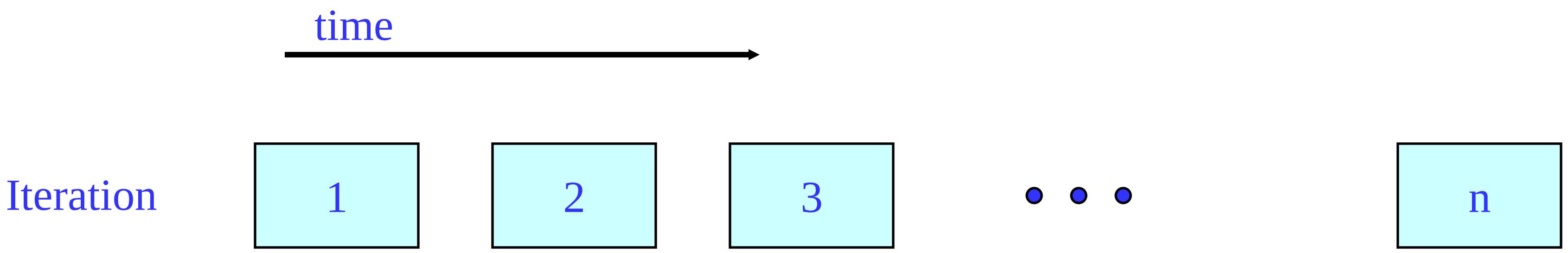
```
for (j=0; j<100; j++)
    b[j] = a[j] * 26
```



```
r1 = _a
r2 = _b
r9 = r1 * 4
```

```
Loop:  1: r3 = load(r1)
       2: r4 = r3 * 26
       3: store (r2, r4)
       4: r1 = r1 + 4
       5: r2 = r2 + 4
       6: p1 = cmpp (r1 < r9)
       7: brct p1 Loop
```

Basic Approach – List Schedule the Loop Body



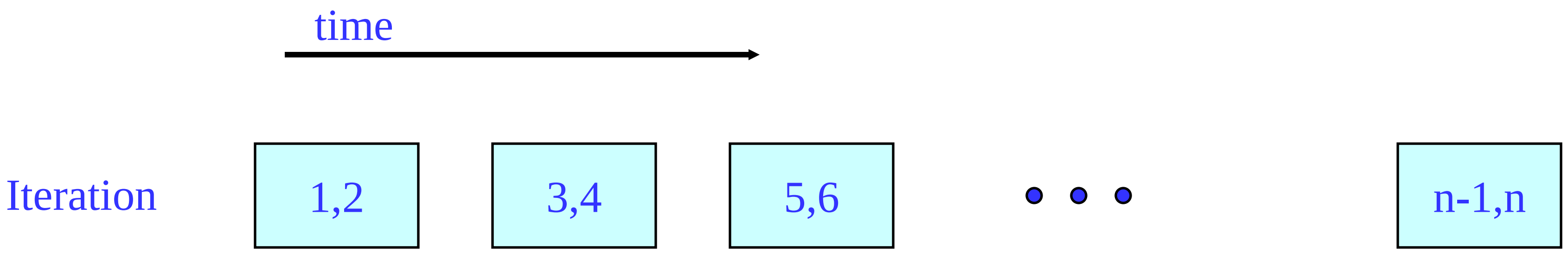
Schedule each iteration
resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

```
1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop
```

time	ops
0	1, 4
1	6
2	2
3	-
4	-
5	3, 5, 7

Total time = 6 * n

Unroll Then Schedule Larger Body



Schedule each iteration
resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, cmpp = 1, mpy=3, ld = 2, st = 1, br = 1

```
1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop
```

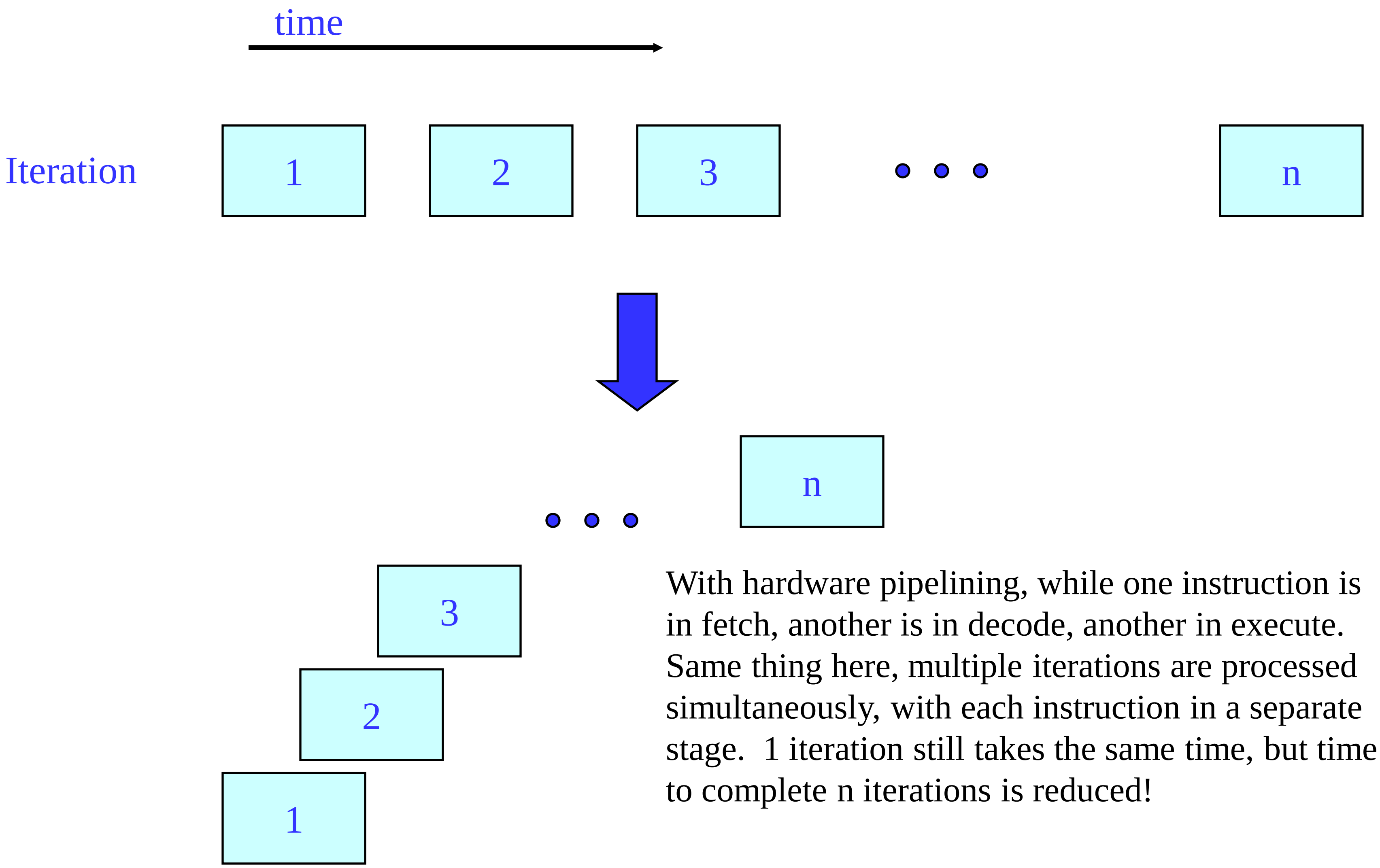
time	ops
0	1, 4
1	1’, 6, 4’
2	2, 6’
3	2’
4	-
5	3, 5, 7
6	3’,5’,7’

Total time = 7 * n/2

Problems With Unrolling

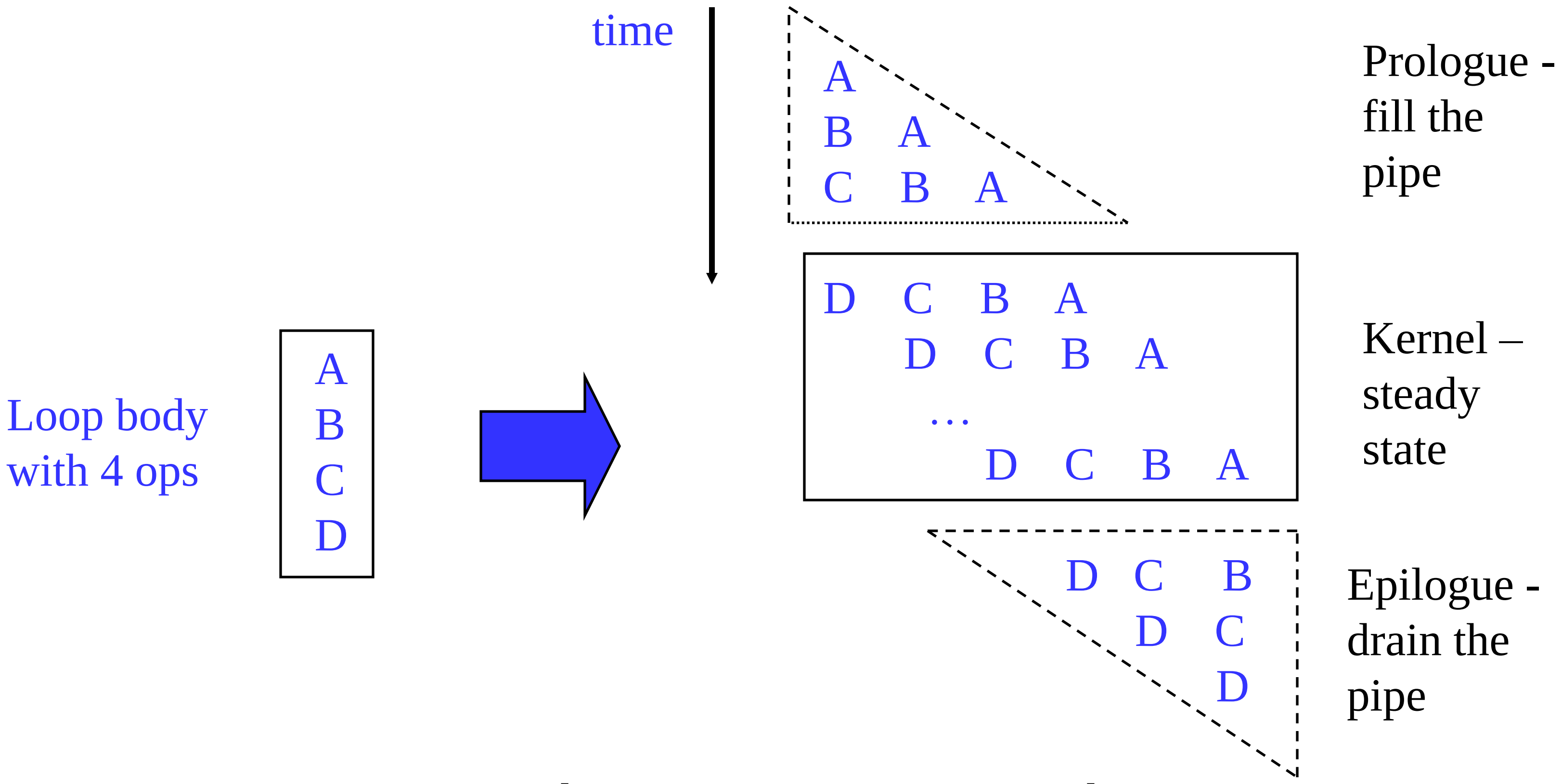
- Code bloat
  - Typical unroll is 4-16x
  - Use profile statistics to only unroll “important” loops
  - But still, code grows fast
- Barrier after across unrolled bodies
  - I.e., for unroll 2, can only overlap iterations 1 and 2, 3 and 4, …
- Does this mean unrolling is bad?
  - No, in some settings its very useful
    - Low trip count
    - Lots of branches in the loop body
  - But, in other settings, there is room for improvement

Overlap Iterations Using Pipelining



With hardware pipelining, while one instruction is in fetch, another is in decode, another in execute. Same thing here, multiple iterations are processed simultaneously, with each instruction in a separate stage. 1 iteration still takes the same time, but time to complete n iterations is reduced!

A Software Pipeline



Steady state: 4 iterations executed simultaneously, 1 operation from each iteration. Every cycle, an iteration starts and finishes when the pipe is full.

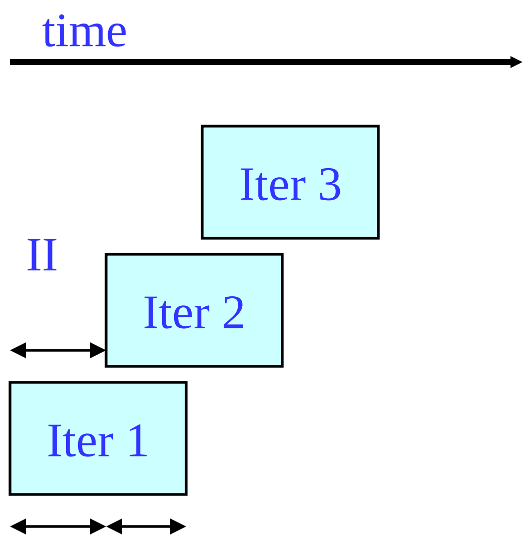
Creating Software Pipelines

- Lots of software pipelining techniques out there
- Modulo scheduling
  - Most widely adopted
  - Practical to implement, yields good results
- Conceptual strategy
  - Unroll the loop completely
  - Then, schedule the code completely with 2 constraints
    - All iteration bodies have identical schedules
    - Each iteration is scheduled to start some fixed number of cycles later than the previous iteration
  - Initiation Interval (II) = fixed delay between the start of successive iterations
  - Given the 2 constraints, the unrolled schedule is repetitive (kernel) except the portion at the beginning (prologue) and end (epilogue)
    - Kernel can be re-rolled to yield a new loop

Creating Software Pipelines (2)

- Create a schedule for 1 iteration of the loop such that when the same schedule is repeated at intervals of II cycles
  - No intra-iteration dependence is violated
  - No inter-iteration dependence is violated
  - No resource conflict arises between operation in same or distinct iterations
- We will start out assuming Itanium-style hardware support, then remove it later
  - Rotating registers
  - Predicates
  - Software pipeline loop branch

Terminology



Initiation Interval (II) = fixed delay between the start of successive iterations

Each iteration can be divided into stages consisting of II cycles each

Number of stages in 1 iteration is termed the stage count (SC)

Takes SC-1 cycles to fill/drain the pipe

To Be Continued …