

EECS 583 – Class 13

Modulo Scheduling

University of Michigan

October 29, 2018

Announcements + Reading Material

- ❖ Project proposals
 - » Due Wednesday, Oct 31, 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 & Ze, cc all your group members
- ❖ Midterm exam
 - » Originally scheduled for Wed Nov 7
 - » Move to Wed Nov 14 in class
 - » More on the content later
- ❖ Today's class reading
 - » “Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops”, B. Rau, MICRO-27, 1994, pp. 63-74.
- ❖ Wed class reading
 - » “Code Generation Schema for Modulo Scheduled Loops”, B. Rau, M. Schlansker, and P. Tirumalai, MICRO-25, Dec. 1992.

Research Paper Presentations

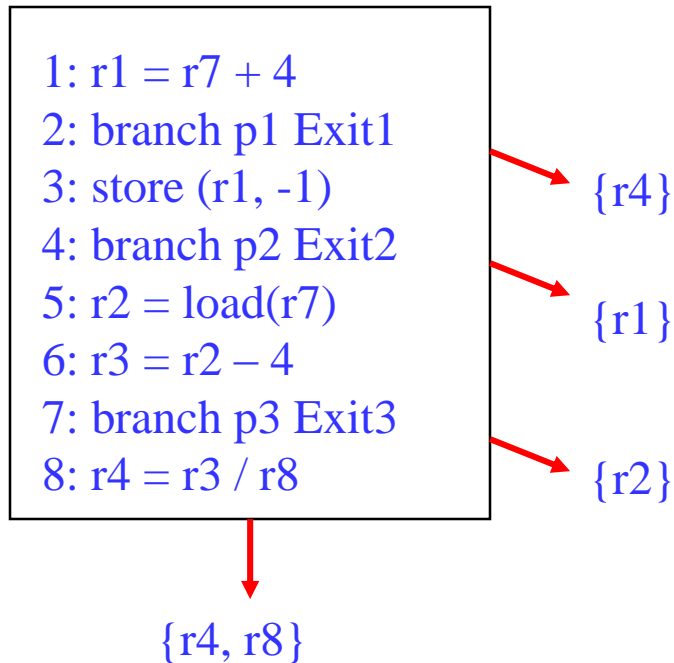
- ❖ Monday Nov 19– Monday Dec 10
 - » Signup for slot next Monday in class or on my door afterwards
- ❖ Each group: 15 min presentation + 5 min Q&A
 - » Tag-team presentation – Divide up as you like but everyone must talk
 - » Max of 16 slides (for the group)
 - » Submit paper pdf 1 week ahead and slides ppt or pdf night before
- ❖ Presentation
 - » Make your own slides
 - » Points to discuss
 - Intro/Motivation – area + problem + why it being solved
 - How the technique works
 - Some results
 - Commentary
 - ◆ What is best about the paper? Why is the idea so awesome? Don't focus on results
 - ◆ What are limitations/weaknesses of the approach (be critical!)

Research Paper Presentations (2)

❖ Audience members

- » Research presentations != skip class, You should be here!
- » Grading + give comments to your peers
 - Class + Ze & I will evaluate each group's presentation and provide feedback
 - Each person will turn in evaluation sheet for the day's presentations
 - Ze & I will anonymize comments and email to each group
 - Be critical, but constructive with your criticisms

Class Problem from Last Time - Solution



1. With general speculation, edges from $2 \rightarrow 5$, $4 \rightarrow 5$, $4 \rightarrow 8$, $7 \rightarrow 8$ can be removed

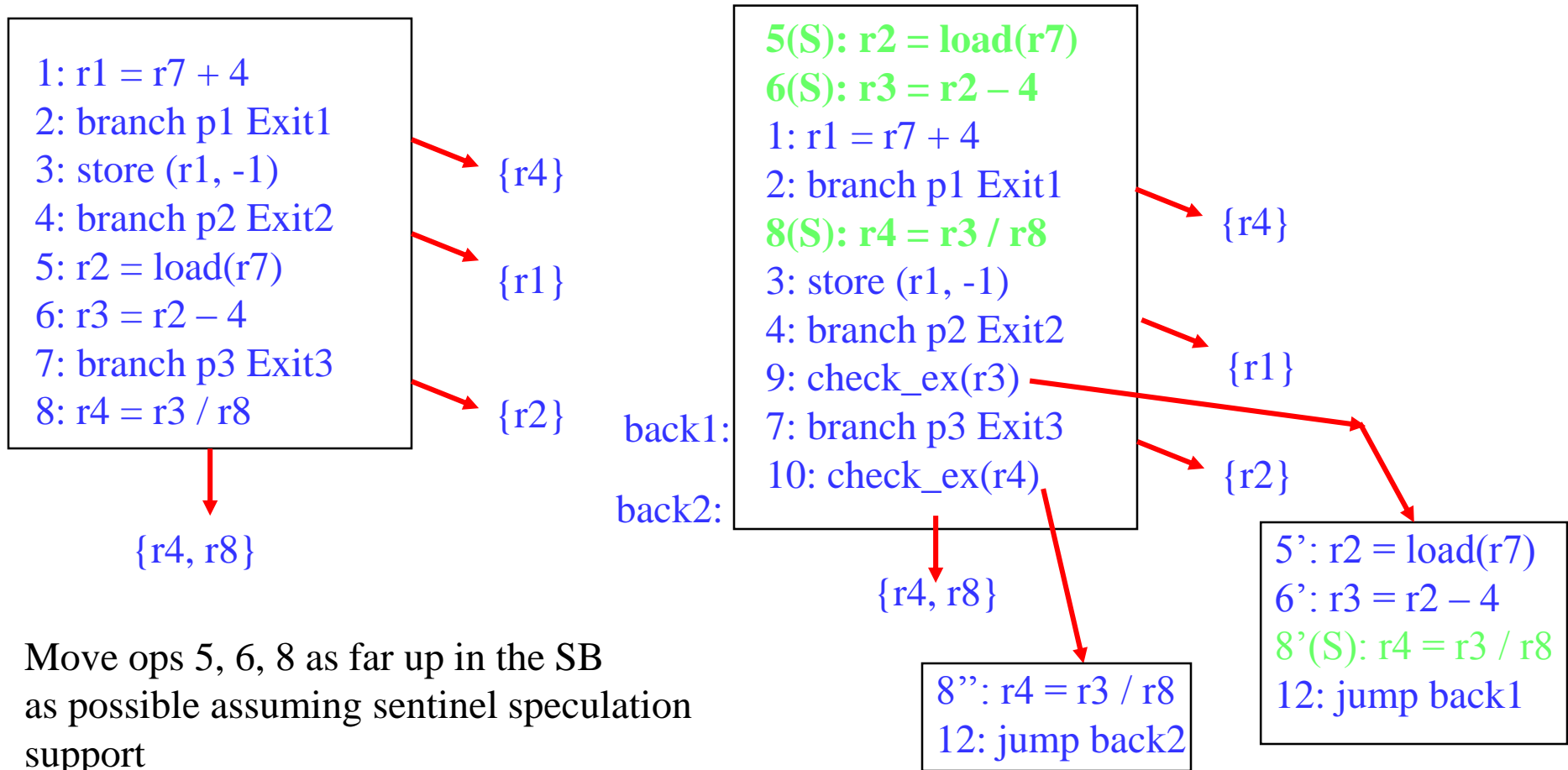
2. With further renaming, the edge from $2 \rightarrow 8$ can be removed.

Note, the edge from $2 \rightarrow 3$ cannot be removed since we conservatively do not allow stores to speculate.

Note2, you do not need general speculation to remove edges from $2 \rightarrow 6$ and $4 \rightarrow 6$ since integer subtract never causes exception.

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?

Class Problem from Last Time – Solution



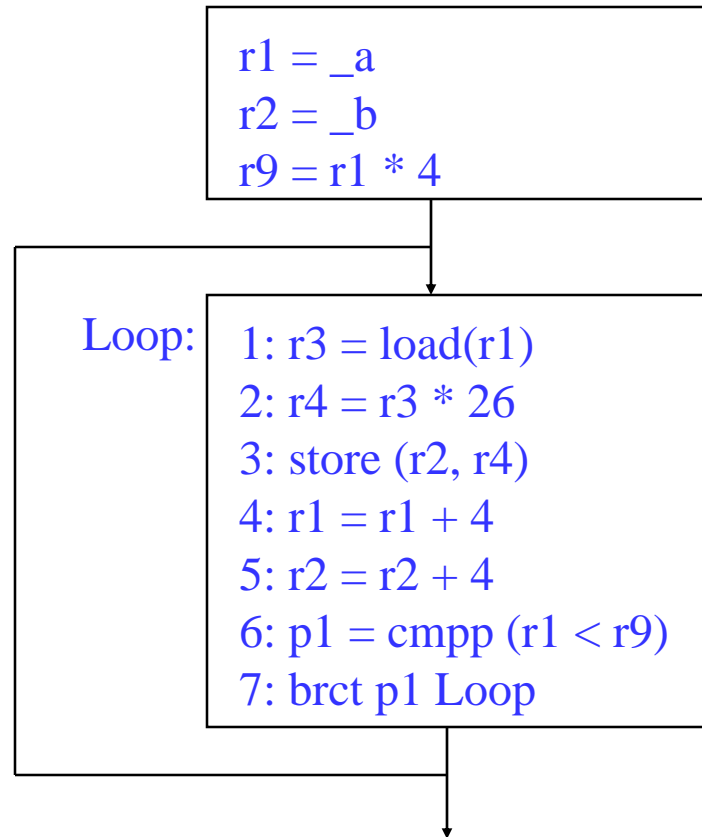
1. Move ops 5, 6, 8 as far up in the SB as possible assuming sentinel speculation support
2. Insert the necessary checks and recovery code (assume ld, st, and div can cause exceptions)

Change Focus to Scheduling Loops

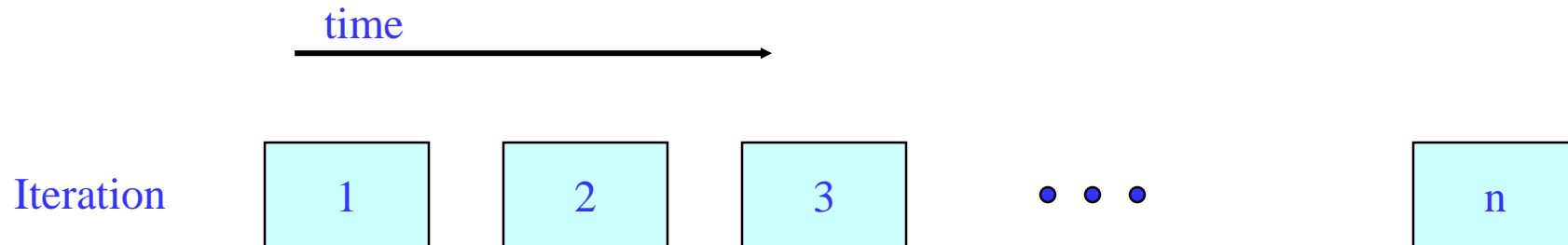
Most of program execution
time is spent in loops

Problem: How do we achieve
compact schedules for loops

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



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

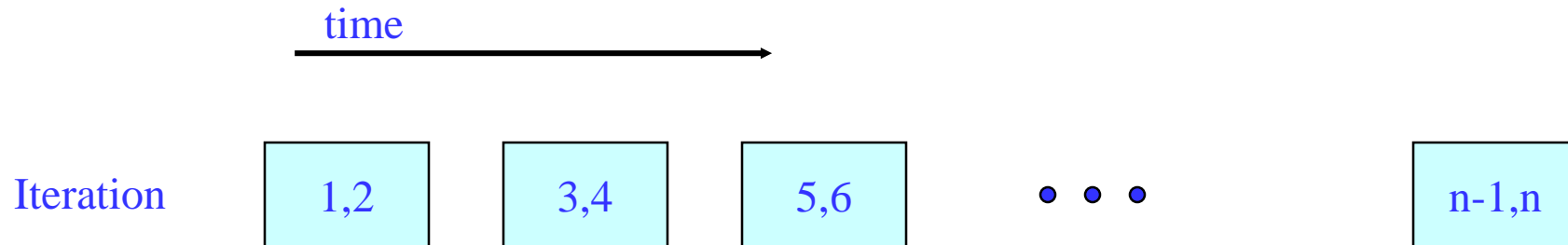
6: p1 = cmpp (r1 < r9)

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
    
```

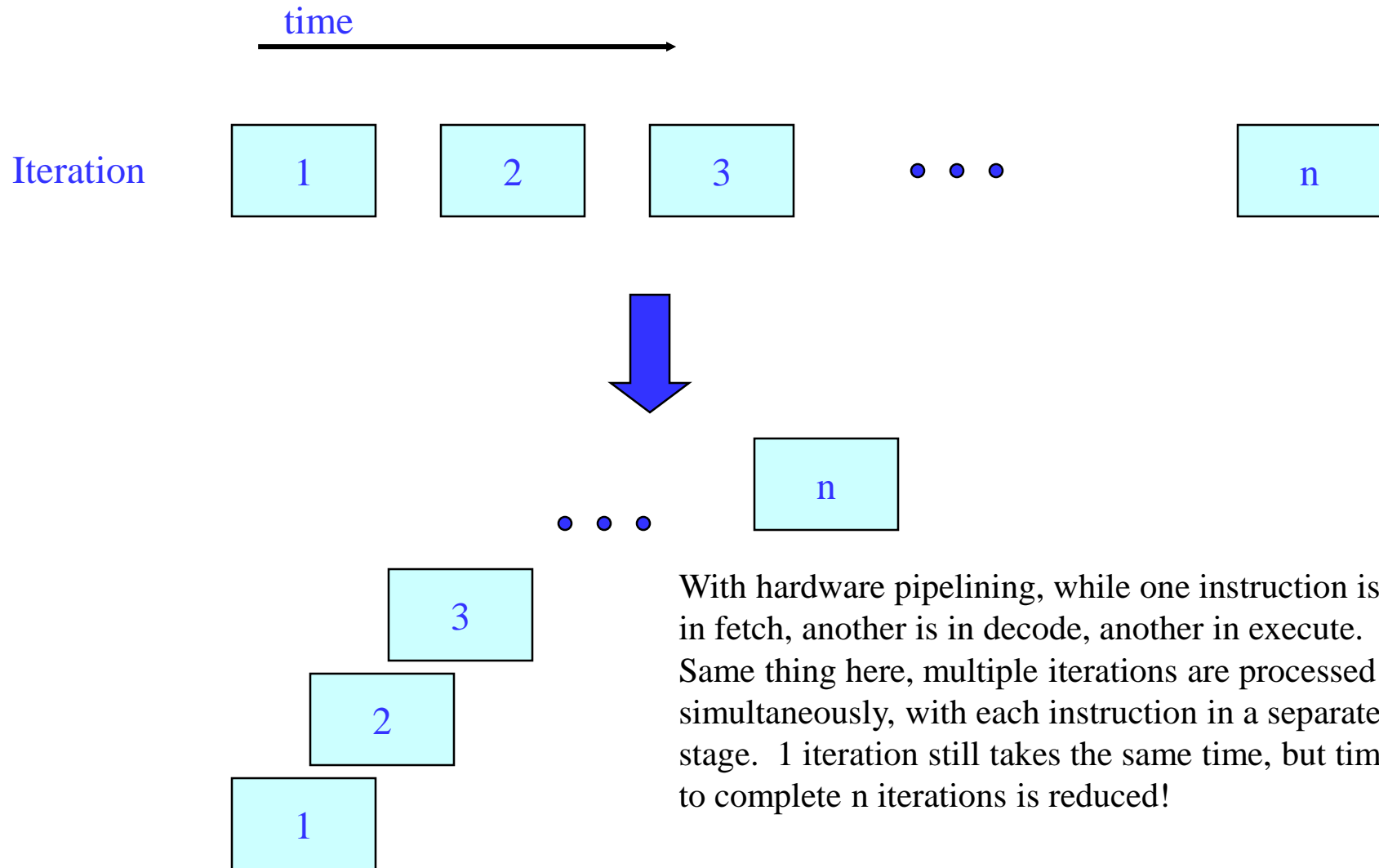
time	ops
0	1, 4
1	1', 6, 4'
2	2, 6'
3	2'
4	-
5	3, 5, 7
6	3', 5', 7'

Total time = 7 * n/2

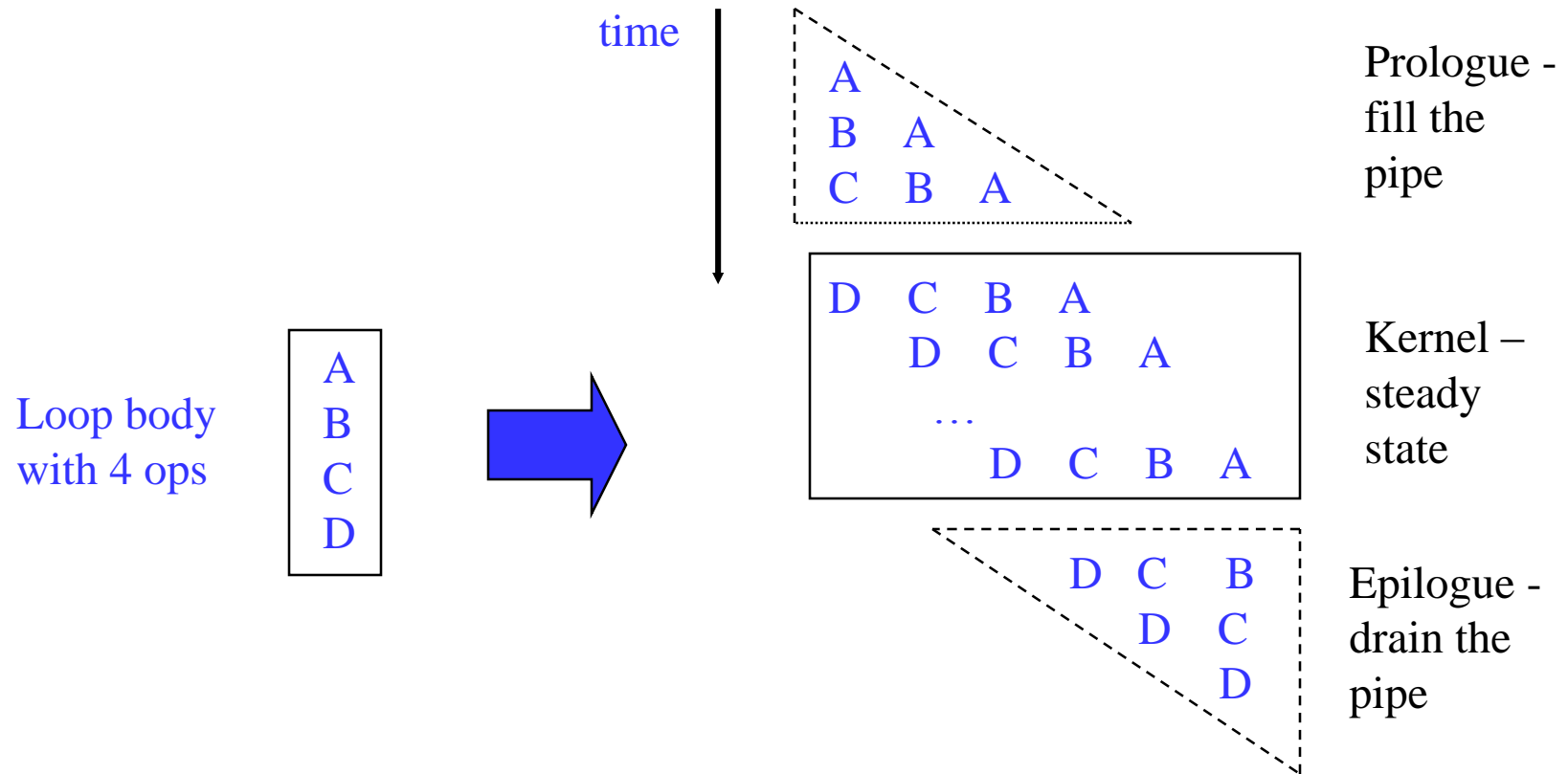
Problems With Unrolling

- ❖ 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



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

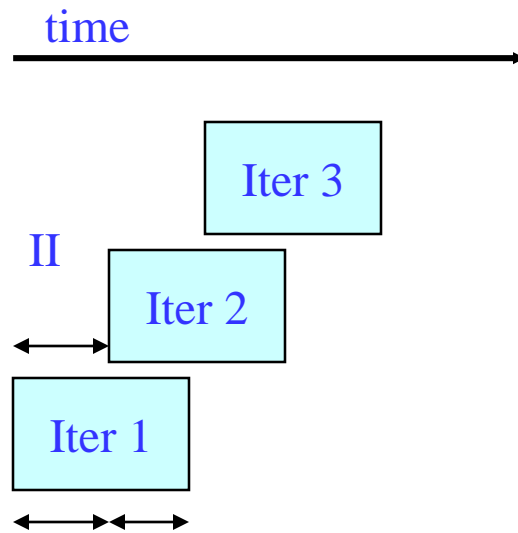
- ❖ Lots of software pipelining techniques out there
- ❖ Modulo scheduling
 - » Most widely adopted
 - » Practical to implement, yields good results
- ❖ Conceptual strategy
 - » Unroll the loop completely
 - » Then, schedule the code completely with 2 constraints
 - All iteration bodies have identical 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 Π 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

Resource Usage Legality

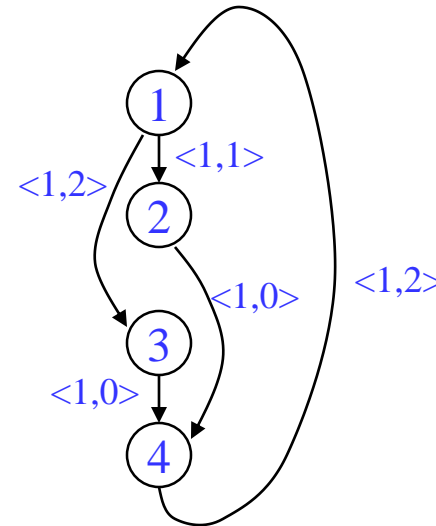
- ❖ Need to guarantee that
 - » No resource is used at 2 points in time that are separated by an interval which is a multiple of Π
 - » I.E., within a single iteration, the same resource is never used more than 1x at the same time modulo Π
 - » Known as modulo constraint, where the name modulo scheduling comes from
 - » Modulo reservation table solves this problem
 - To schedule an op at time T needing resource R
 - ◆ The entry for R at $T \bmod \Pi$ must be free
 - Mark busy at $T \bmod \Pi$ if schedule

$$\Pi = 3$$

	alu1	alu2	mem	bus0	bus1	br
0						
1						
2						

Dependences in a Loop

- ❖ Need worry about 2 kinds
 - » Intra-iteration
 - » Inter-iteration
- ❖ Delay
 - » Minimum time interval between the start of operations
 - » Operation read/write times
- ❖ Distance
 - » Number of iterations separating the 2 operations involved
 - » Distance of 0 means intra-iteration
- ❖ Recurrence manifests itself as a circuit in the dependence graph



Edges annotated with tuple

$\langle \text{delay}, \text{distance} \rangle$

Dynamic Single Assignment (DSA) Form


Impossible to overlap iterations because each iteration writes to the same register. So, we'll have to remove the anti and output dependences.

Virtual rotating registers

- * Each register is an infinite push down array (Expanded virtual reg or EVR)
- * Write to top element, but can reference any element
- * Remap operation slides everything down $\rightarrow r[n]$ changes to $r[n+1]$

A program is in DSA form if the same virtual register (EVR element) is never assigned to more than 1x on any dynamic execution path

```
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
```


DSA
conversion

```
1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
6: p1[-1] = cmpp (r1[-1] < r9)
  remap r1, r2, r3, r4, p1
7: brct p1[-1] Loop
```

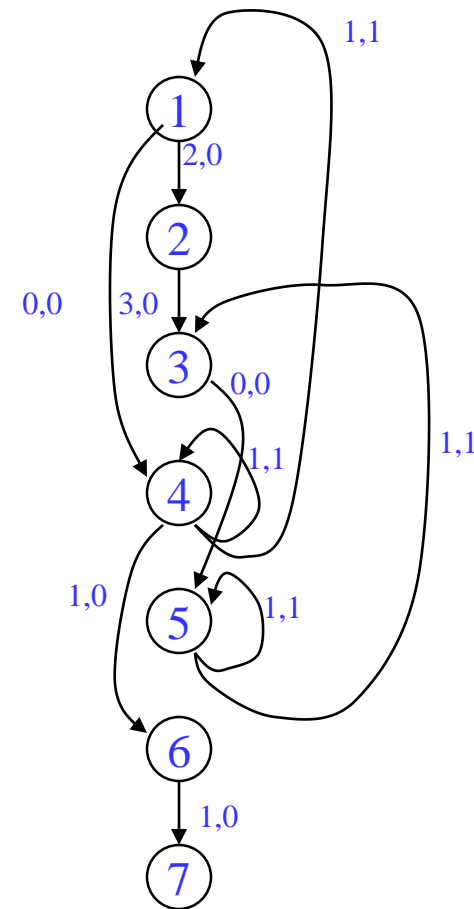
Physical Realization of EVRs

- ❖ EVR may contain an unlimited number values
 - » But, only a finite contiguous set of elements of an EVR are ever live at any point in time
 - » These must be given physical registers
- ❖ Conventional register file
 - » Remaps are essentially copies, so each EVR is realized by a set of physical registers and copies are inserted
- ❖ Rotating registers
 - » Direct support for EVRs
 - » No copies needed
 - » File “rotated” after each loop iteration is completed

Loop Dependence Example

```
1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
6: p1[-1] = cmpp (r1[-1] < r9)
  remap r1, r2, r3, r4, p1
7: brct p1[-1] Loop
```

In DSA form, there are no inter-iteration anti or output dependences!



<delay, distance>

Class Problem

Latencies: ld = 2, st = 1, add = 1, cmpp = 1, br = 1

```
1: r1[-1] = load(r2[0])
2: r3[-1] = r1[1] - r1[2]
3: store (r3[-1], r2[0])
4: r2[-1] = r2[0] + 4
5: p1[-1] = cmpp (r2[-1] < 100)
   remap r1, r2, r3
6: brct p1[-1] Loop
```

Draw the dependence graph
showing both intra and inter
iteration dependences

Minimum Initiation Interval (MII)

- ❖ Remember, II = number of cycles between the start of successive iterations
- ❖ Modulo scheduling requires a candidate II be selected before scheduling is attempted
 - » Try candidate II , see if it works
 - » If not, increase by 1, try again repeating until successful
- ❖ MII is a lower bound on the II
 - » $MII = \text{Max}(\text{ResMII}, \text{RecMII})$
 - » ResMII = resource constrained MII
 - Resource usage requirements of 1 iteration
 - » RecMII = recurrence constrained MII
 - Latency of the circuits in the dependence graph

ResMII

Concept: If there were no dependences between the operations, what is the the shortest possible schedule?

Simple resource model

A processor has a set of resources R . For each resource r in R there is $\text{count}(r)$ specifying the number of identical copies

$$\text{ResMII} = \text{MAX}_{\text{for all } r \text{ in } R} (\text{uses}(r) / \text{count}(r))$$

$\text{uses}(r)$ = number of times the resource is used in 1 iteration

In reality its more complex than this because operations can have multiple alternatives (different choices for resources it could be assigned to), but we will ignore this for now

ResMII Example

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

ALU: used by 2, 4, 5, 6

→ 4 ops / 2 units = 2

Mem: used by 1, 3

→ 2 ops / 1 unit = 2

Br: used by 7

→ 1 op / 1 unit = 1

ResMII = MAX(2,2,1) = 2

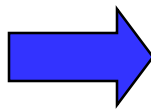
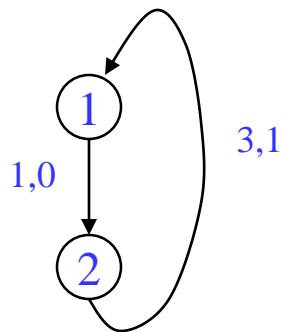
RecMII

Approach: Enumerate all irredundant elementary circuits in the dependence graph

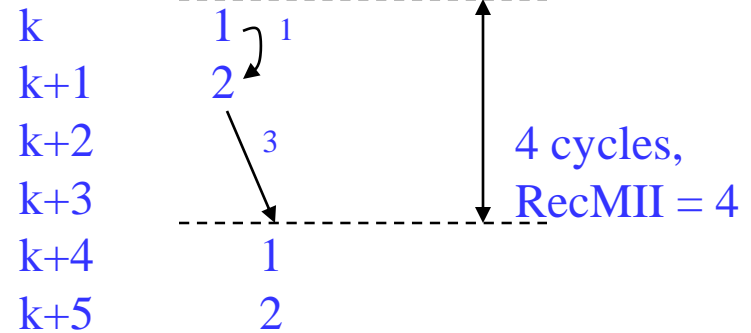
$$\text{RecMII} = \text{MAX}_{\text{for all } c \text{ in } C} (\text{delay}(c) / \text{distance}(c))$$

$\text{delay}(c)$ = total latency in dependence cycle c (sum of delays)

$\text{distance}(c)$ = total iteration distance of cycle c (sum of distances)



cycle



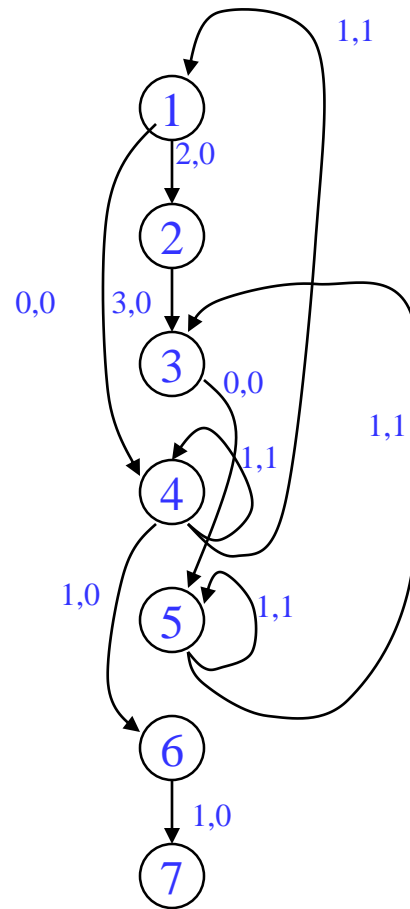
$$\text{delay}(c) = 1 + 3 = 4$$

$$\text{distance}(c) = 0 + 1 = 1$$

$$\text{RecMII} = 4/1 = 4$$

RecMII Example

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



<delay, distance>

4 → 4: 1 / 1 = 1
5 → 5: 1 / 1 = 1
4 → 1 → 4: 1 / 1 = 1
5 → 3 → 5: 1 / 1 = 1

RecMII = MAX(1,1,1,1) = 1

Then,

MII = MAX(ResMII, RecMII)

MII = MAX(2,1) = 2

Class Problem

Latencies: ld = 2, st = 1, add = 1, cmpp = 1, br = 1

Resources: 1 ALU, 1 MEM, 1 BR

```
1: r1[-1] = load(r2[0])
2: r3[-1] = r1[1] - r1[2]
3: store (r3[-1], r2[0])
4: r2[-1] = r2[0] + 4
5: p1[-1] = cmpp (r2[-1] < 100)
   remap r1, r2, r3
6: brct p1[-1] Loop
```

Calculate RecMII, ResMII, and MII

Modulo Scheduling Process

- ❖ Use list scheduling but we need a few twists
 - » Π is predetermined – starts at $M\Pi$, then is incremented
 - » Cyclic dependences complicate matters
 - Estart/Priority/etc.
 - Consumer scheduled before producer is considered
 - ◆ There is a window where something can be scheduled!
 - » Guarantee the repeating pattern
- ❖ 2 constraints enforced on the schedule
 - » Each iteration begin exactly Π cycles after the previous one
 - » Each time an operation is scheduled in 1 iteration, it is tentatively scheduled in subsequent iterations at intervals of Π
 - MRT used for this

Priority Function

Height-based priority worked well for acyclic scheduling, makes sense that it will work for loops as well

Acyclic:

$$\text{Height}(X) = \begin{cases} 0, & \text{if } X \text{ has no successors} \\ \text{MAX}_{\text{for all } Y = \text{succ}(X)} ((\text{Height}(Y) + \text{Delay}(X, Y)), & \text{otherwise} \end{cases}$$

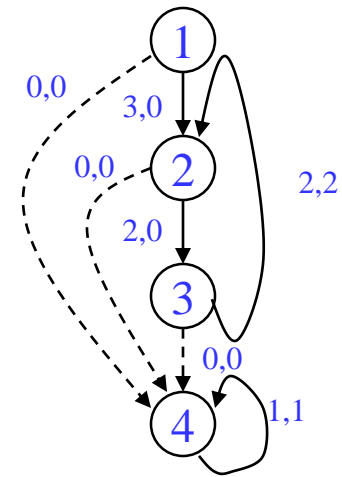
Cyclic:

$$\text{HeightR}(X) = \begin{cases} 0, & \text{if } X \text{ has no successors} \\ \text{MAX}_{\text{for all } Y = \text{succ}(X)} ((\text{HeightR}(Y) + \text{EffDelay}(X, Y)), & \text{otherwise} \end{cases}$$

$$\text{EffDelay}(X, Y) = \text{Delay}(X, Y) - \Pi * \text{Distance}(X, Y)$$

Calculating Height

1. Insert pseudo edges from all nodes to branch with latency = 0, distance = 0 (dotted edges)
2. Compute Π , For this example assume $\Pi = 2$
3. HeightR(4) =
4. HeightR(3) =
5. HeightR(2) =
6. HeightR(1) =



The Scheduling Window

With cyclic scheduling, not all the predecessors may be scheduled, so a more flexible earliest schedule time is:

$$E(Y) = \underset{\text{for all } X = \text{pred}(Y)}{\text{MAX}} \begin{cases} 0, & \text{if } X \text{ is not scheduled} \\ \text{MAX}(0, \text{SchedTime}(X) + \text{EffDelay}(X, Y)), & \text{otherwise} \end{cases}$$

$$\text{where } \text{EffDelay}(X, Y) = \text{Delay}(X, Y) - \Pi * \text{Distance}(X, Y)$$

Every Π cycles a new loop iteration will be initialized, thus every Π cycles the pattern will repeat. Thus, you only have to look in a window of size Π , if the operation cannot be scheduled there, then it cannot be scheduled.

$$\text{Latest schedule time}(Y) = L(Y) = E(Y) + \Pi - 1$$

To Be Continued
