EECS 583 – Class 13
Software Pipelining

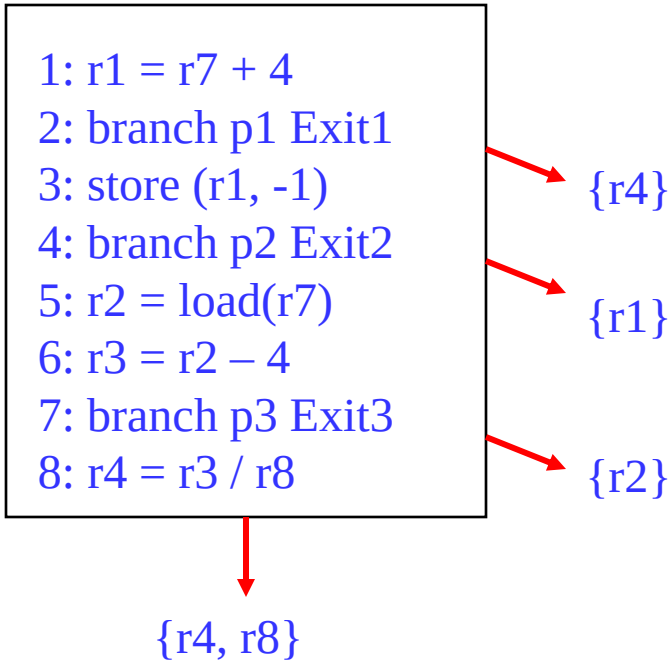
University of Michigan

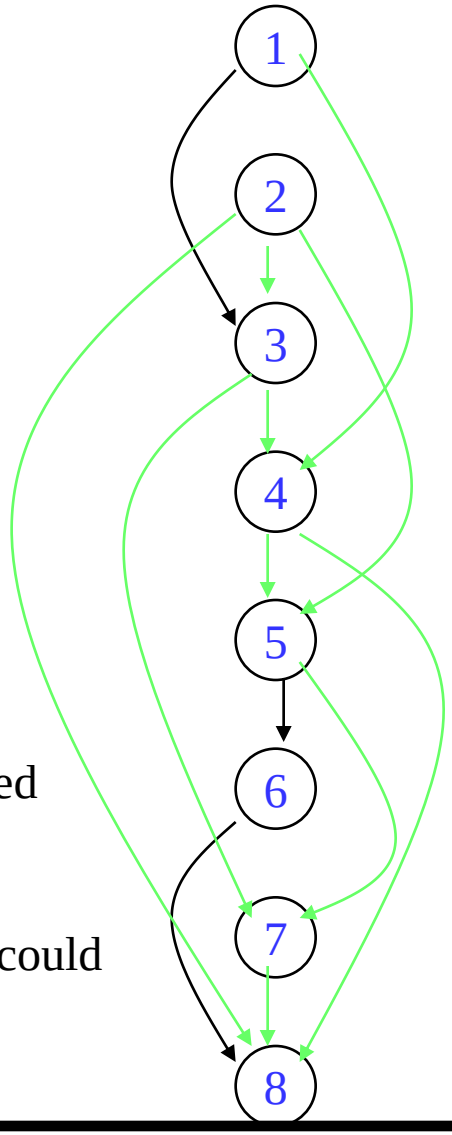
October 24, 2011

Announcements + Reading Material

- No class on Wednesday (reserved for project proposals)
- Each group needs to signup for a 15 min slot this week
  - Signup sheet on my door (4633 CSE)
  - Slots on Tues, Wednes, Thurs and Fri
  - Informal class project proposal discussion
- Homework 2 deadline
  - Today at midnight, or tomorrow midnight if you have not used your late day
  - Daya will have office hours today 3-5pm if you are stuck
- 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.

Class Problem from Last Time





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?

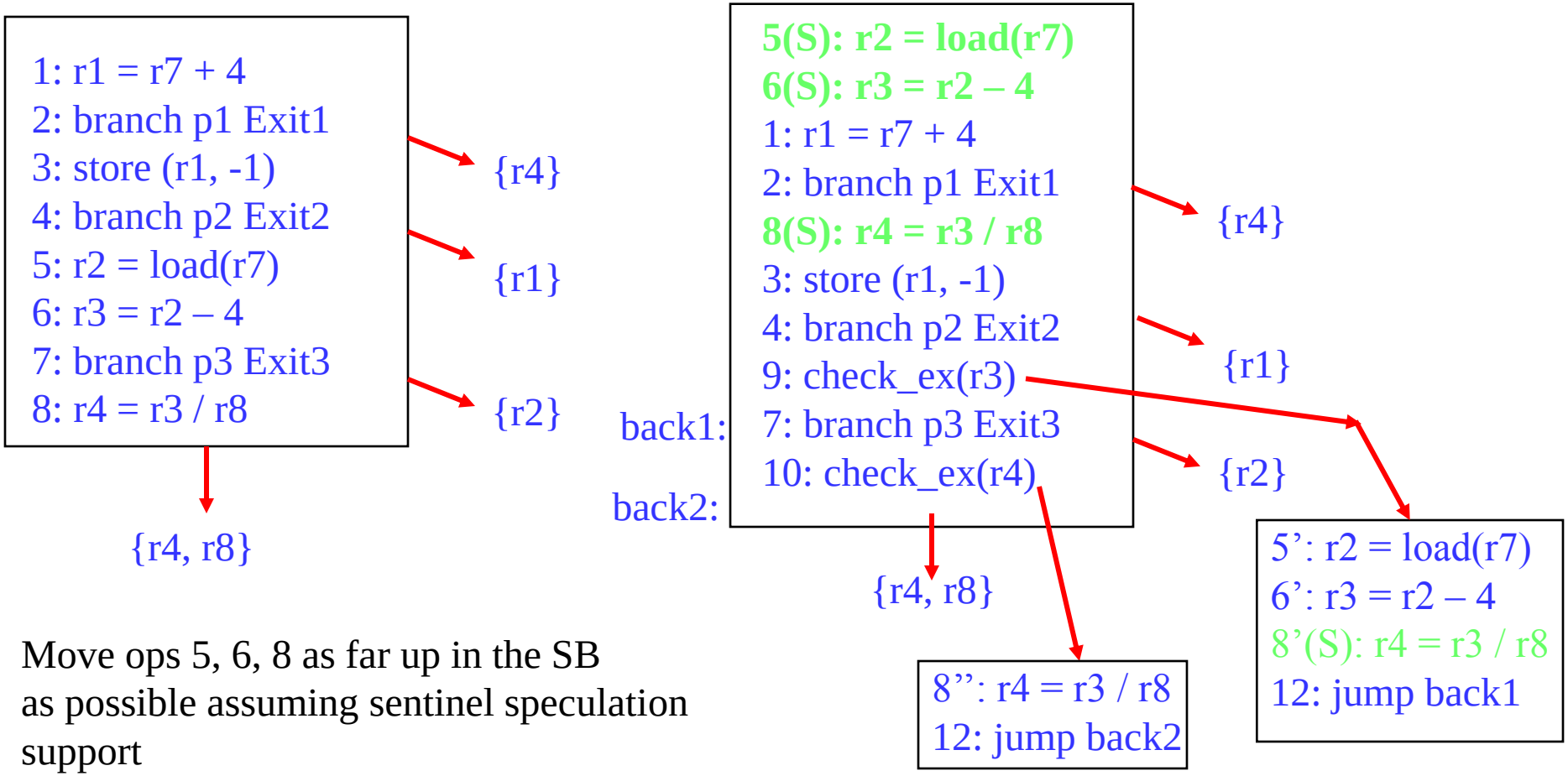
Edges not drawn:
2 → 4, 2→ 7, 4→ 7

There is no edge from 3 to 5 if you assume 32-bit load/store instructions since r1 and r7 are 4 different..

Answer 1:
2 → 5, 4→ 5 since r2 is not live out; 4 → 8, 7→ 8 since r4 is not live out, but 2→ 8 must remain;

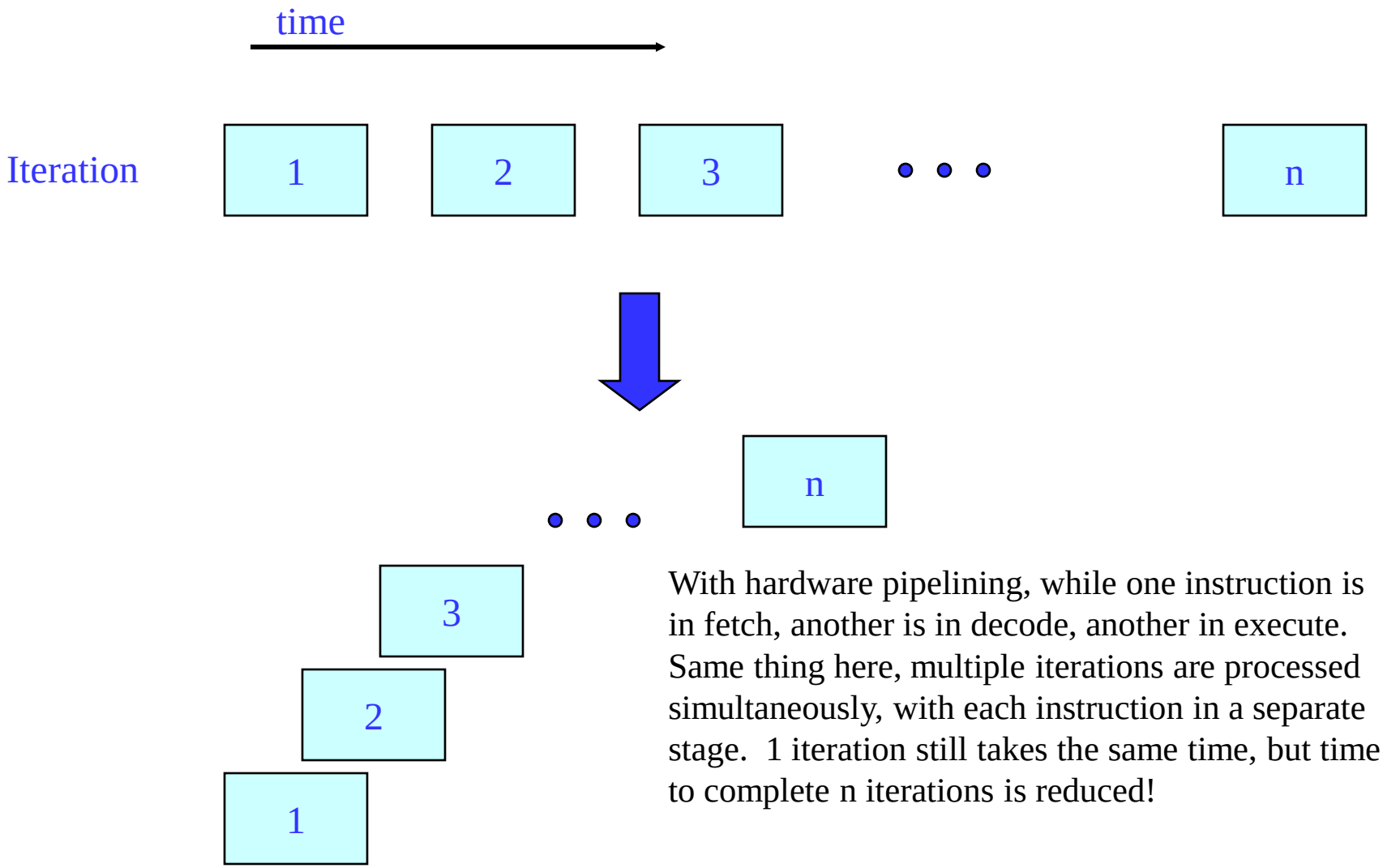
Answer 2:
2 → 8

Class Problem from Last Time



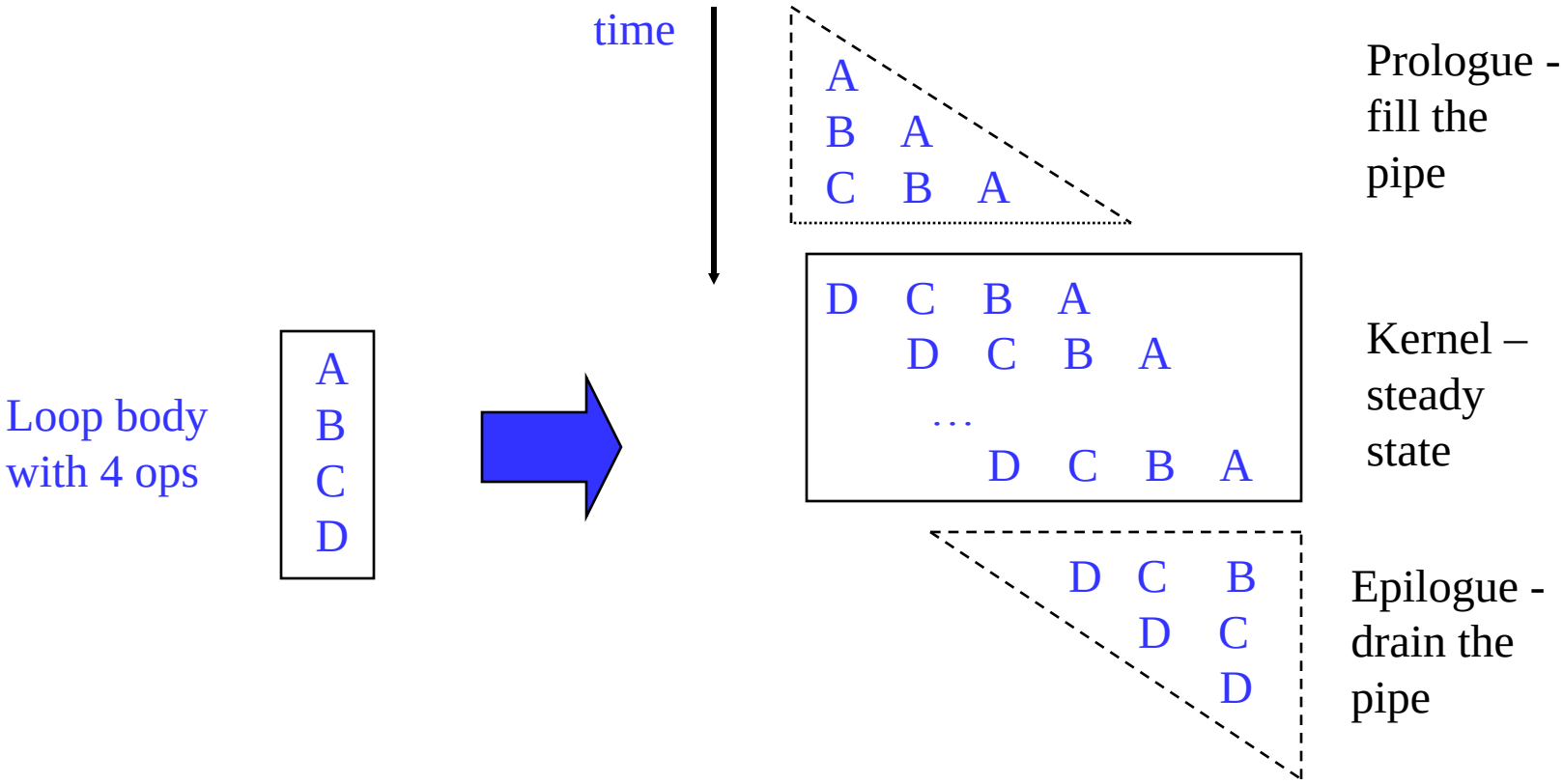
1. Move ops 5, 6, 8 as far up in the SB as possible assuming sentinel speculation support
2. Insert the necessary checks and recovery code (assume ld, st, and div can cause exceptions)

Review: 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!

Review: 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.

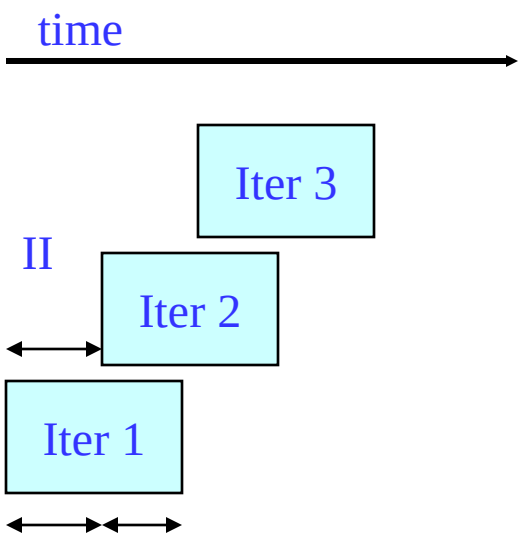
Creating Software Pipelines

- 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

Resource Usage Legality

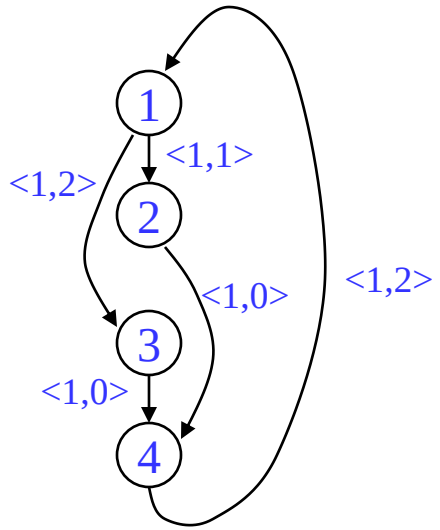
- Need to guarantee that
  - No resource is used at 2 points in time that are separated by an interval which is a multiple of II
  - I.E., within a single iteration, the same resource is never used more than 1x at the same time modulo II
  - Known as modulo constraint, where the name modulo scheduling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

II = 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

<delay, distance>

Dynamic Single Assignment (DSA) Form

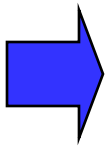
Impossible to overlap iterations because each iteration writes to the same register. So, we'll have to remove the anti and output dependences.

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 → 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
```

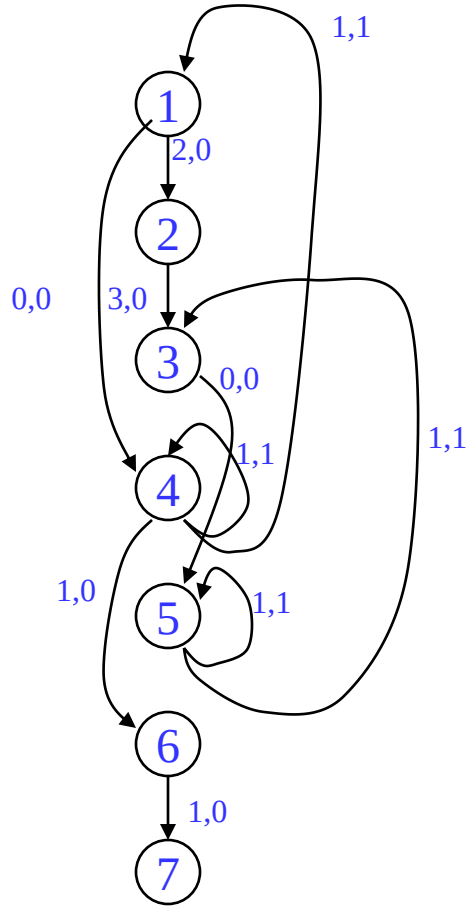
Physical Realization of EVRs

- 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 = Max(ResMII, 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 count(r) specifying the number of identical copies

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

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

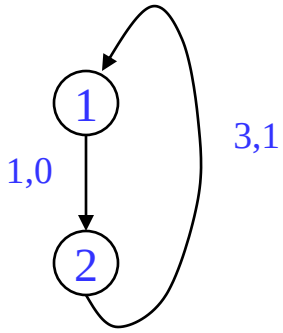
ResMII = MAX(2,2,1) = 2

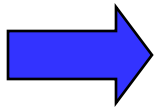
RecMII

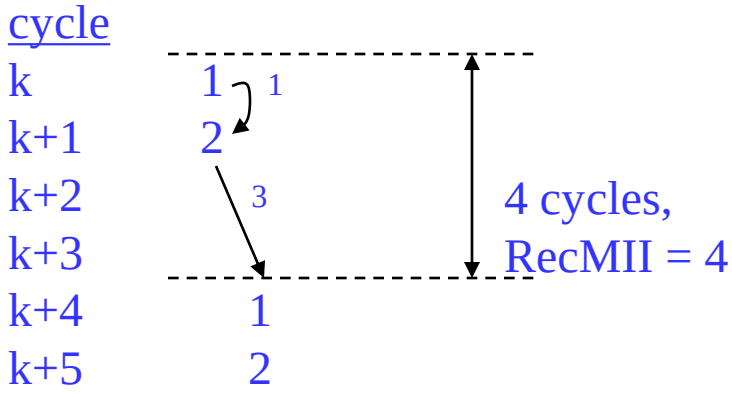
Approach: Enumerate all irredundant elementary circuits in the dependence graph

RecMII = MAX (delay(c) / distance(c))
for all c in C

delay(c) = total latency in dependence cycle c (sum of delays)
distance(c) = total iteration distance of cycle c (sum of distances)



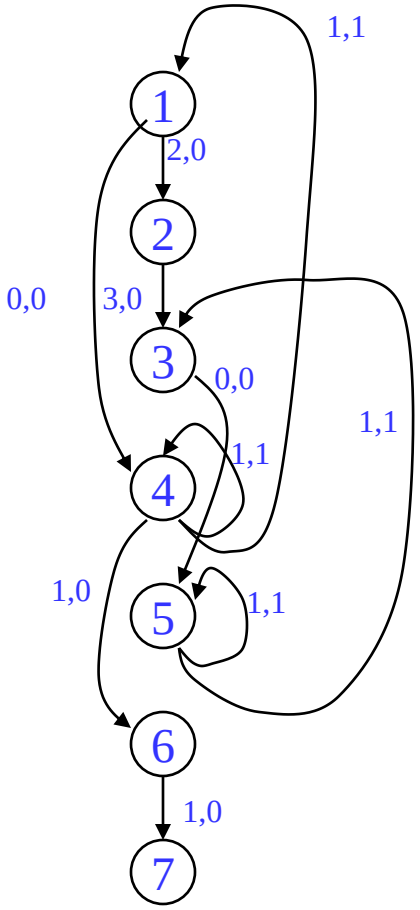




delay(c) = 1 + 3 = 4
distance(c) = 0 + 1 = 1
RecMII = 4/1 = 4

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
  - II is predetermined – starts at MII, 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 II cycles after the previous one
  - Each time an operation is scheduled in 1 iteration, it is tentatively scheduled in subsequent iterations at intervals of II
    - 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:

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

Cyclic:

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

$$EffDelay(X,Y) = Delay(X,Y) - II*Distance(X,Y)$$

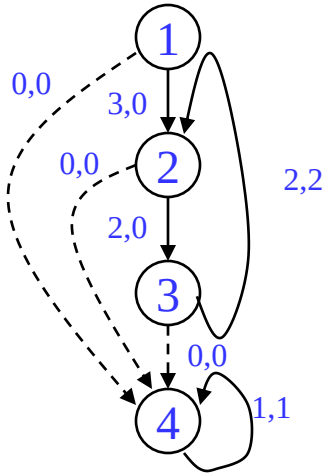
Calculating Height

1. Insert pseudo edges from all nodes to branch with latency = 0, distance = 0 (dotted edges)
2. Compute II, For this example assume II = 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:

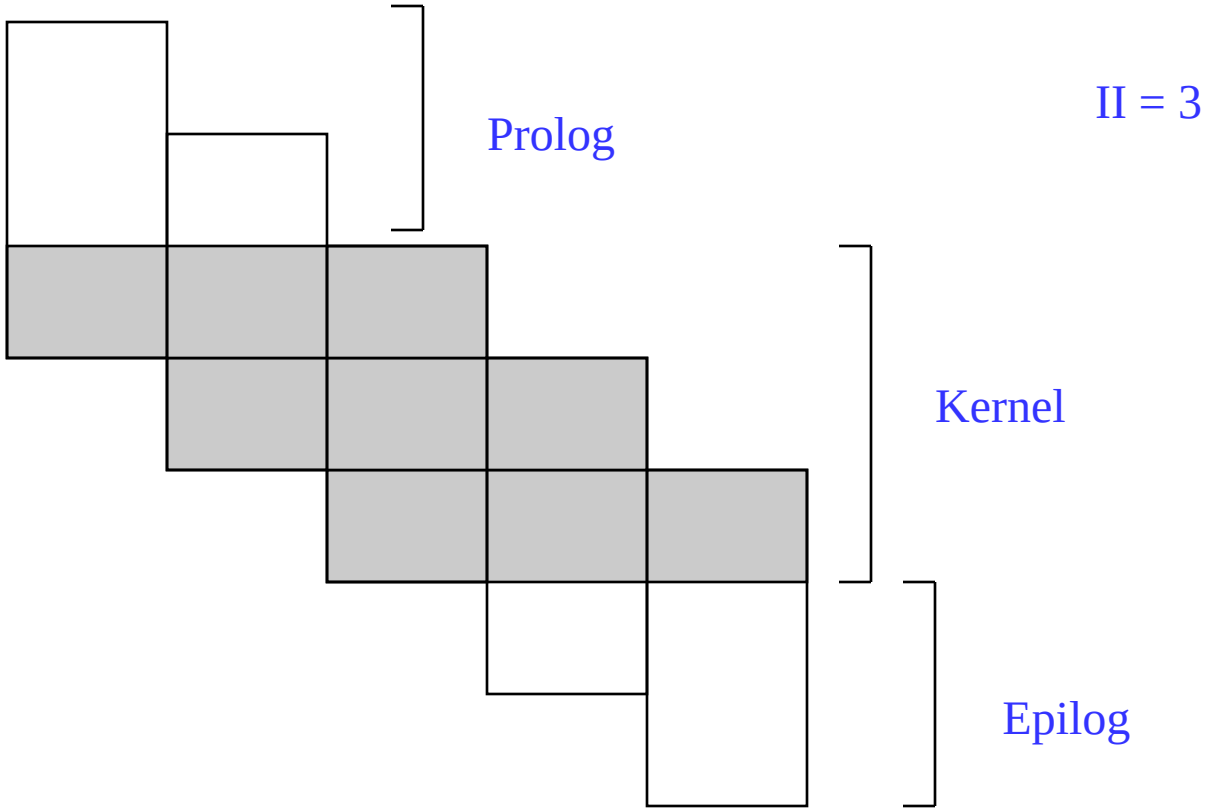
$$E(Y) = \max_{\text{for all } X = \text{pred}(Y)} \begin{cases} 0, \text{ if X is not scheduled} \\ \text{MAX } (0, \text{SchedTime}(X) + \text{EffDelay}(X,Y)), \text{ otherwise} \end{cases}$$

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

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

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

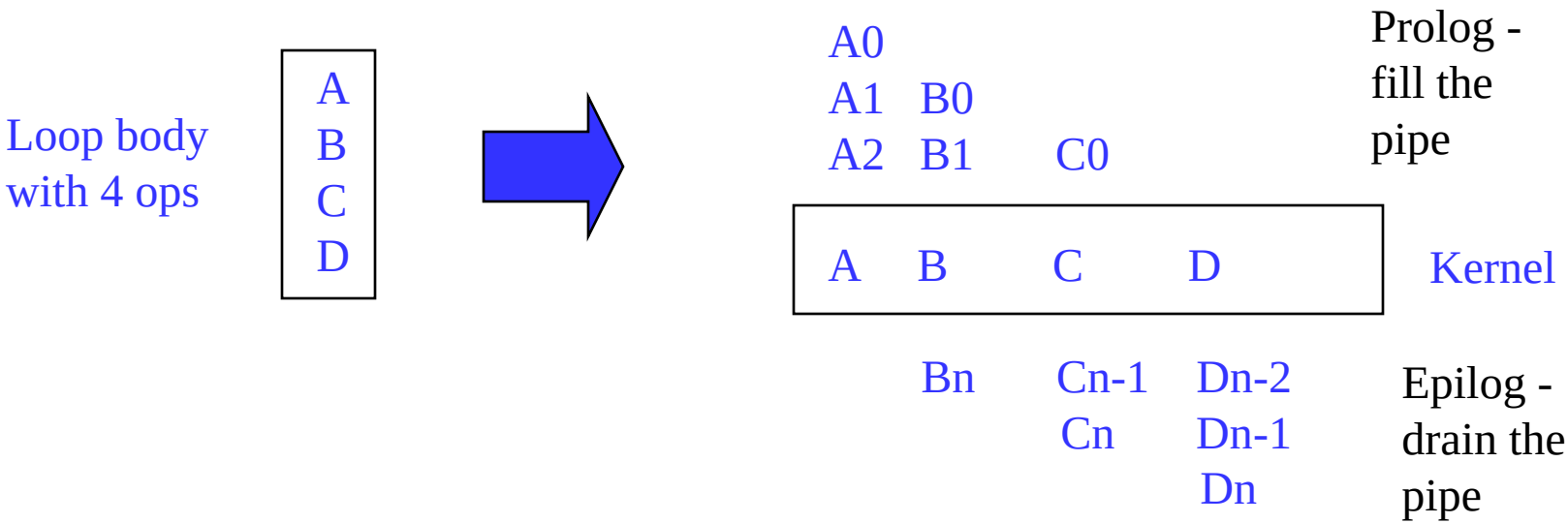
Loop Prolog and Epilog



Only the kernel involves executing full width of operations

Prolog and epilog execute a subset (ramp-up and ramp-down)

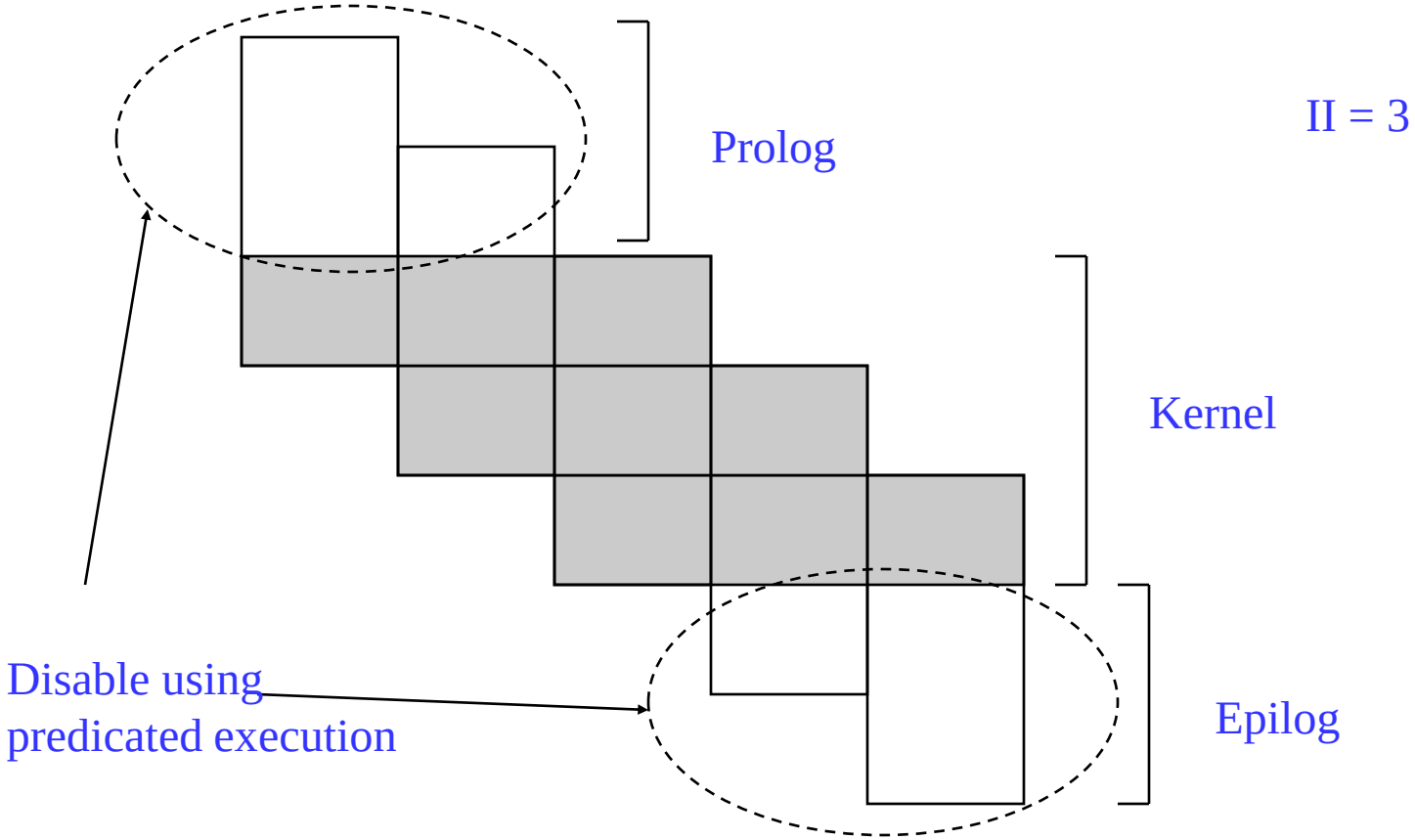
Separate Code for Prolog and Epilog



Generate special code before the loop (preheader) to fill the pipe and special code after the loop to drain the pipe.

Peel off II-1 iterations for the prolog. Complete II-1 iterations in epilog

Removing Prolog/Epilog

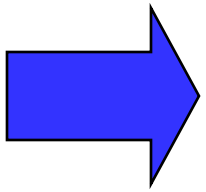


Execute loop kernel on every iteration, but for prolog and epilog selectively disable the appropriate operations to fill/drain the pipeline

Kernel-only Code Using Rotating Predicates

A0
A1 B0
A2 B1 C0

A	B	C	D



A if P[0] B if P[1] C if P[2] D if P[3]

Bn Cn-1 Dn-2
Cn Dn-1
Dn

P referred to as the staging predicate

P[0]	P[1]	P[2]	P[3]
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
…			
0	1	1	1
0	0	1	1
0	0	0	1

A	-	-	-
A	B	-	-
A	B	C	-
A	B	C	D
…			
-	B	C	D
-	-	C	D
-	-	-	D

Modulo Scheduling Architectural Support

- Loop requiring N iterations
  - Will take N + (S – 1) where S is the number of stages
- 2 special registers created
  - LC: loop counter (holds N)
  - ESC: epilog stage counter (holds S)
- Software pipeline branch operations
  - Initialize LC = N, ESC = S in loop preheader
  - All rotating predicates are cleared
  - BRF – software pipeline loopback branch
    - While LC > 0, decrement LC and RRB, P[0] = 1, branch to top of loop
      - This occurs for prolog and kernel
    - If LC = 0, then while ESC > 0, decrement RRB and write a 0 into P[0], and branch to the top of the loop
      - This occurs for the epilog

Execution History With LC/ESC

LC = 3, ESC = 3 /* Remember 0 relative!! */
Clear all rotating predicates
P[0] = 1

```
A if P[0];   B if P[1];   C if P[2];  D if P[3];  P[0] = BRF;
```

LC	ESC	P[0]	P[1]	P[2]	P[3]				
3	3	1	0	0	0	A			
2	3	1	1	0	0	A	B		
1	3	1	1	1	0	A	B	C	
0	3	1	1	1	1	A	B	C	D
0	2	0	1	1	1	-	B	C	D
0	1	0	0	1	1	-	-	C	D
0	0	0	0	0	1	-	-	-	D

4 iterations, 4 stages, II = 1, Note 4 + 4 –1 iterations of kernel executed