Topic 12: Cyclic Instruction Scheduling

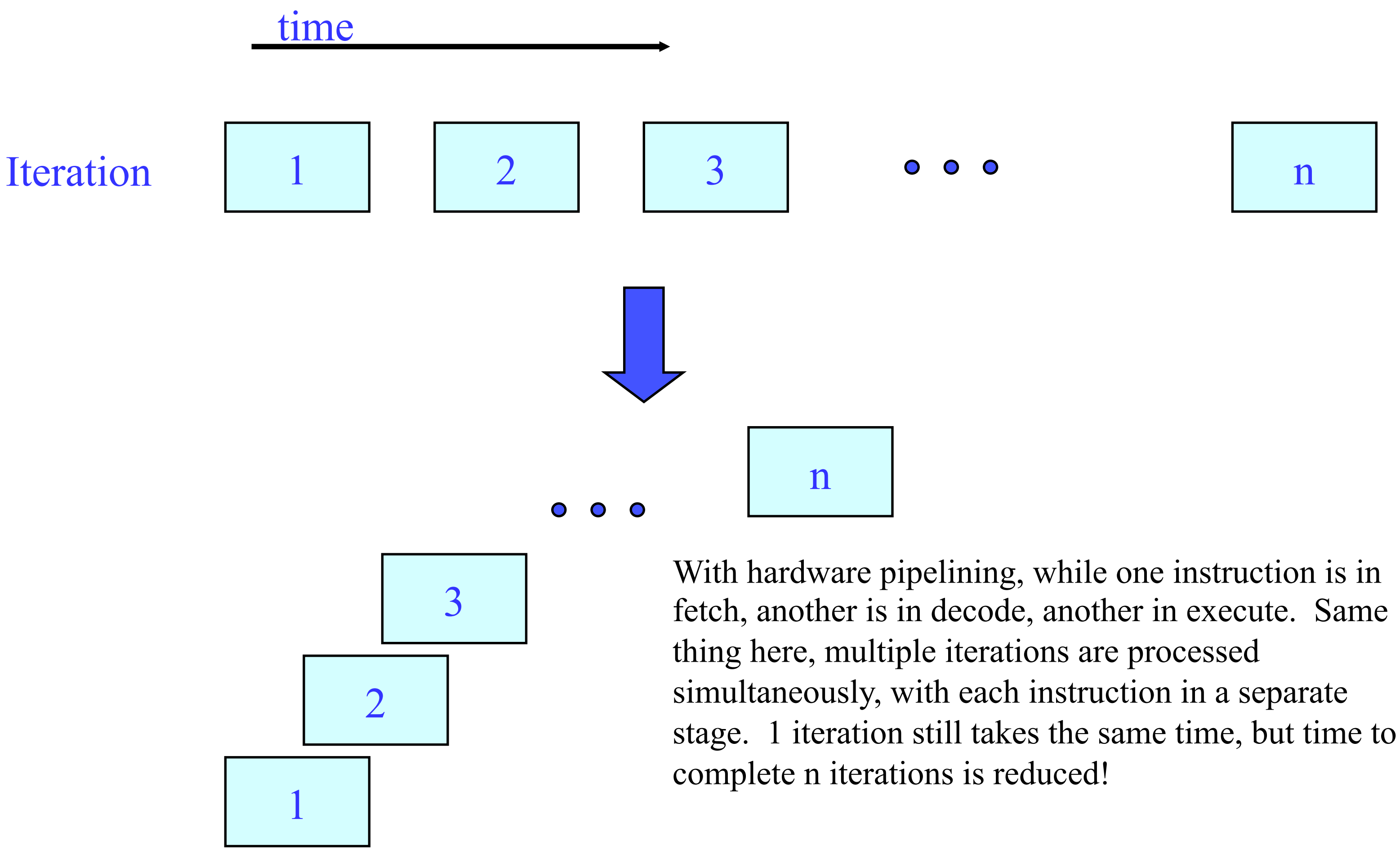
COS 320

Compiling Techniques

Princeton University
Spring 2015

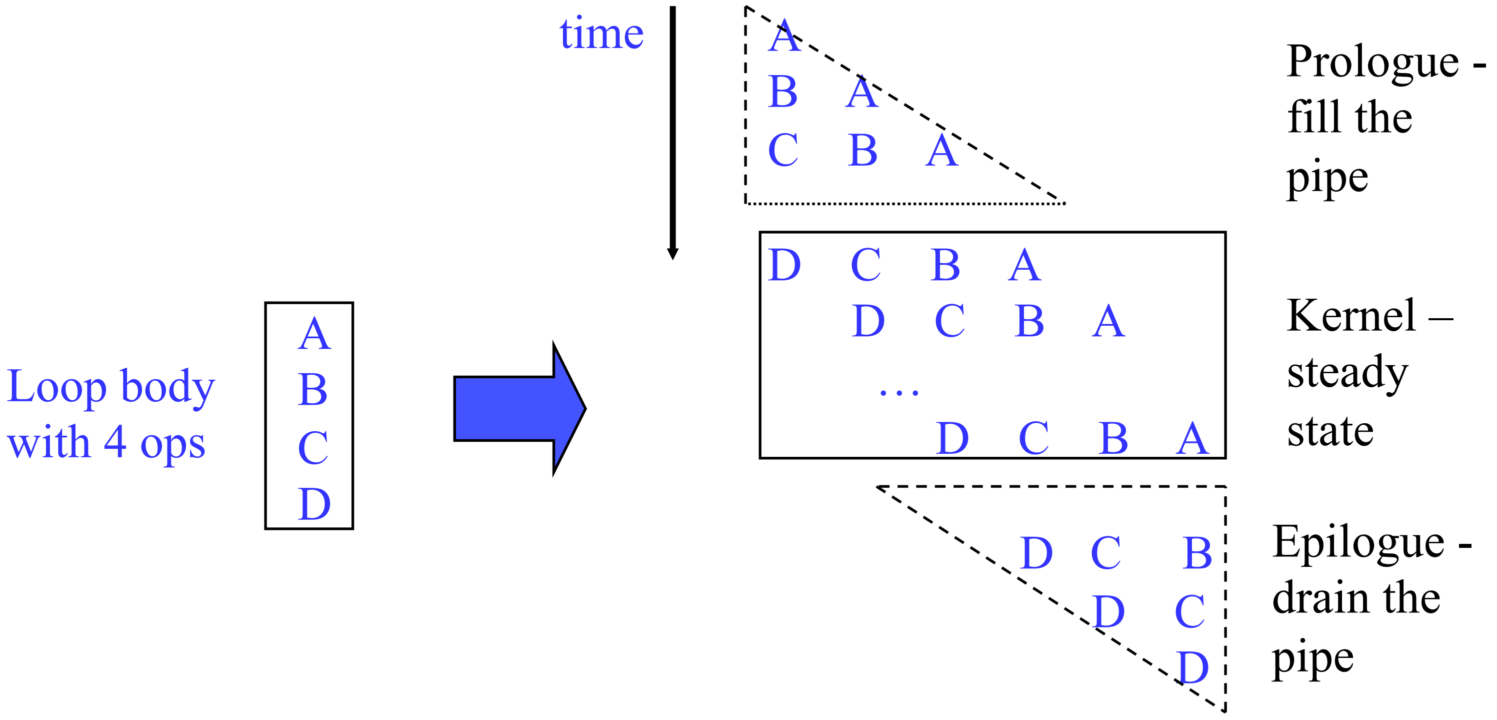
Prof. David August

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.

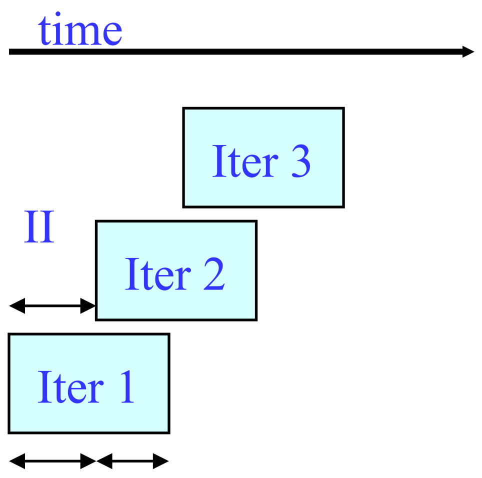
Creating Software Pipelines

- 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 special hardware support for the discussion (e.g. IA-64)
  - Rotating registers
  - Predicates
  - Brtop

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

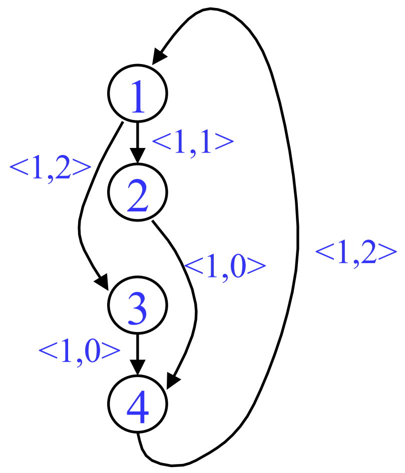
- Need to guarantee that
  - No resource is used at 2 points in time that are separated by an interval which is a multiple of II
  - Within a single iteration, the same resource is never used more than 1x at the same time modulo II
  - Known as modulo constraint, where the name modulo scheduling comes 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

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. Remove the anti and output dependences.

Rotating register (virtual for now)

- * Each register is an infinite push down array (<u>Expanded virtual reg or EVR</u>)
- * Write to top element, but can reference any element
- * Remake 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
```

$\Rightarrow$ 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
```

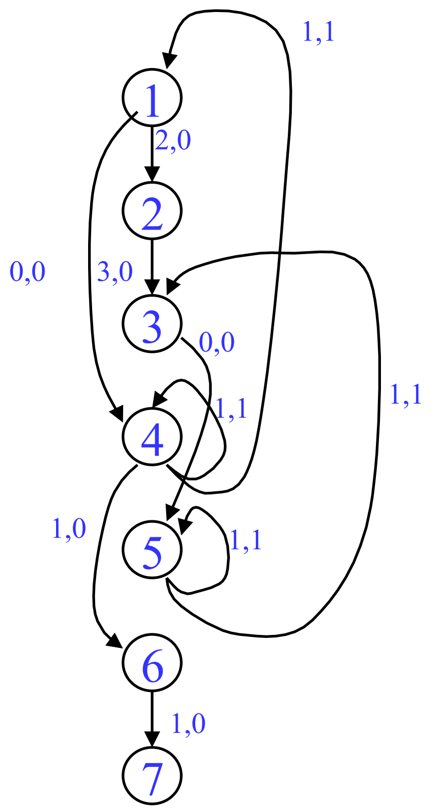
Physical Realization of EVRs

- 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>

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

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

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)

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

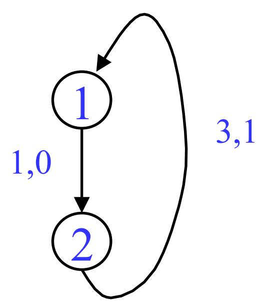
ResMII = MAX(2,2,1) = 2

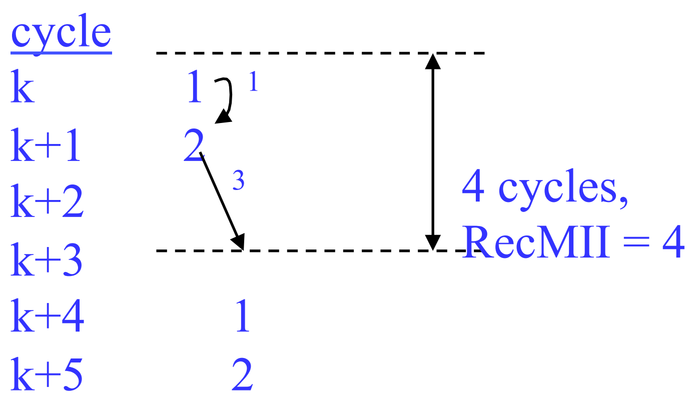
RecMII

Approach: Enumerate all irredundant elementary circuits in the dependence graph

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

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

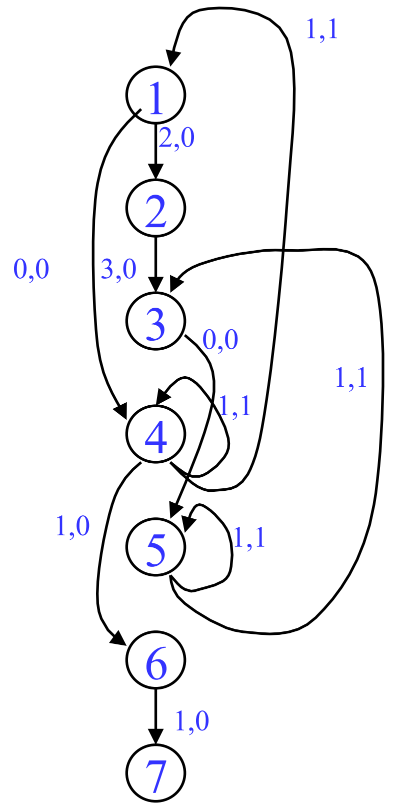




delay(c) = 1 + 3 = 4
distance(c) = 0 + 1 = 1
RecMII = 4/1 = 4

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>

$4 \rightarrow 4$: 1 / 1 = 1
$5 \rightarrow 5$: 1 / 1 = 1
$4 \rightarrow 1 \rightarrow 4$: 1 / 1 = 1
$5 \rightarrow 3 \rightarrow 5$: 1 / 1 = 1

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

Then,

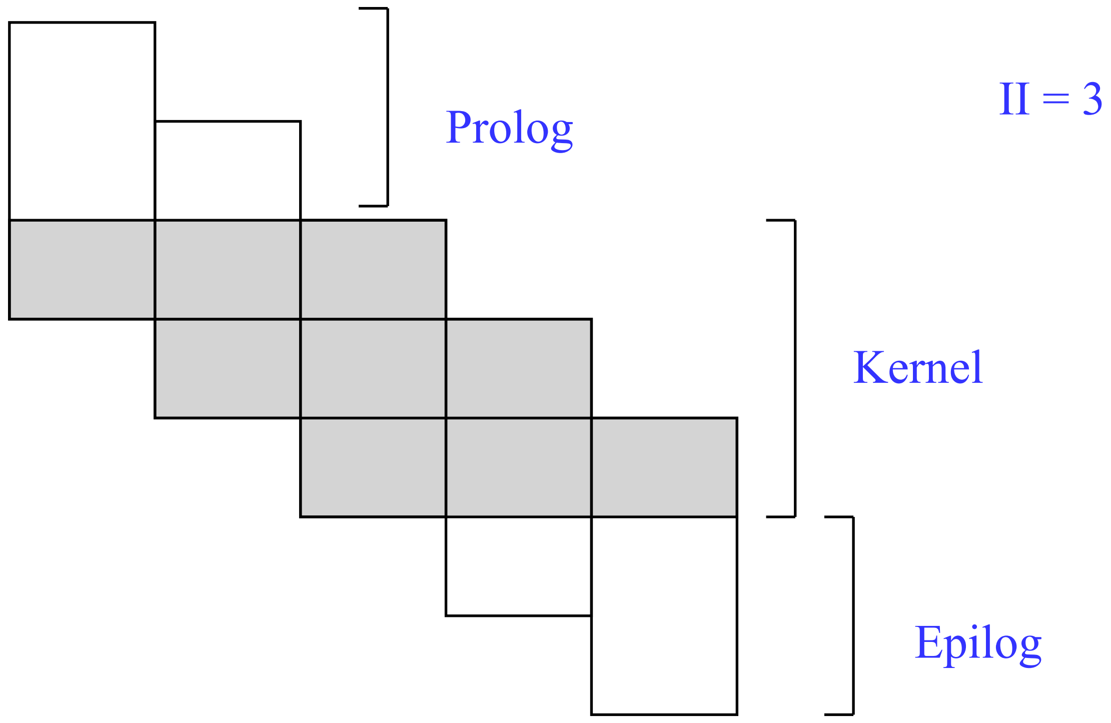
MII = MAX(ResMII, RecMII)
MII = MAX(2,1) = 2

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
    - 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

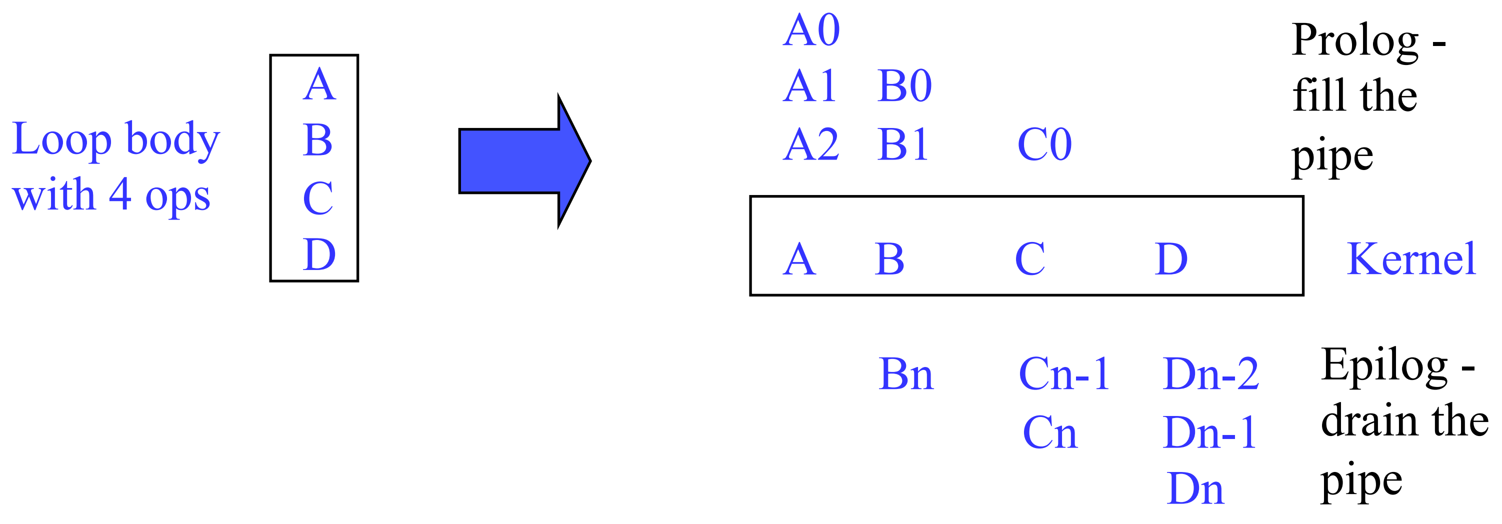
Loop Prolog and Epilog



Only the kernel involves executing full width of operations

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

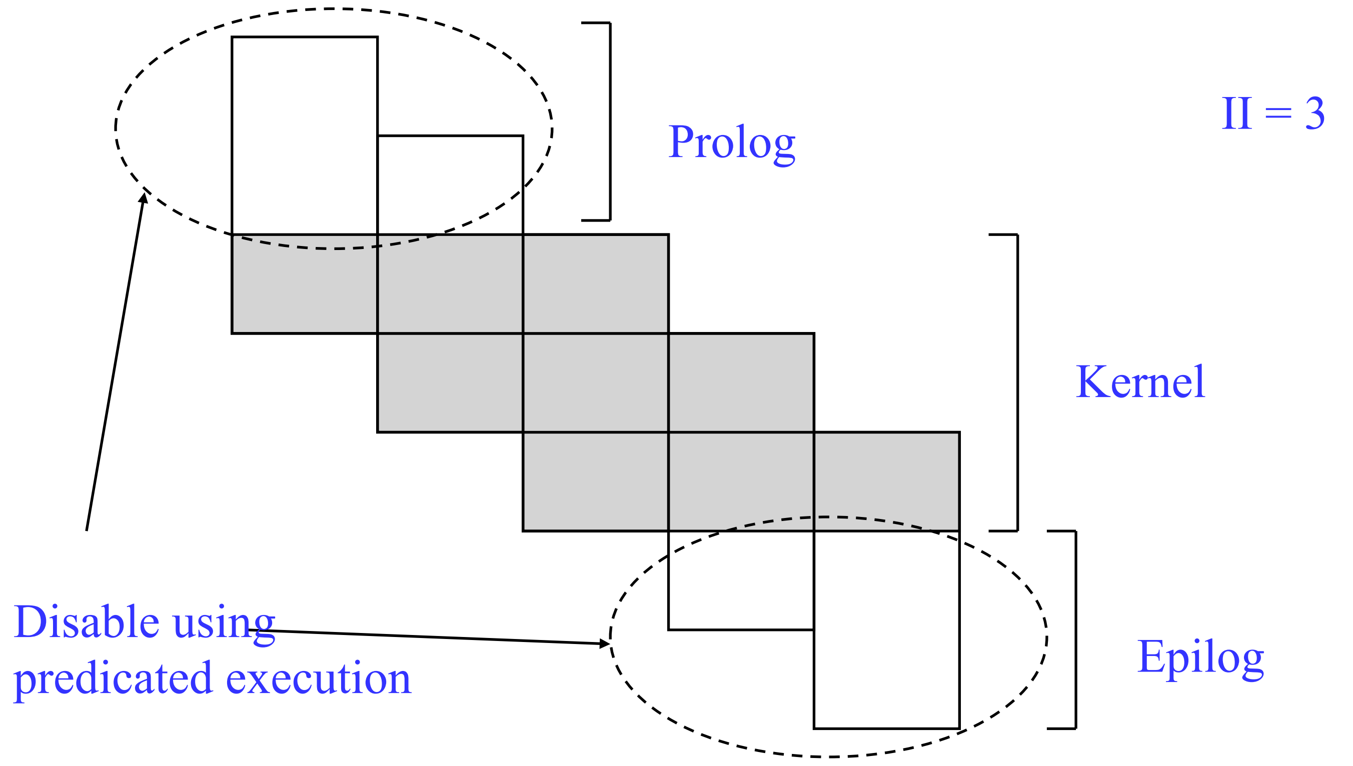
Separate Code for Prolog and Epilog



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

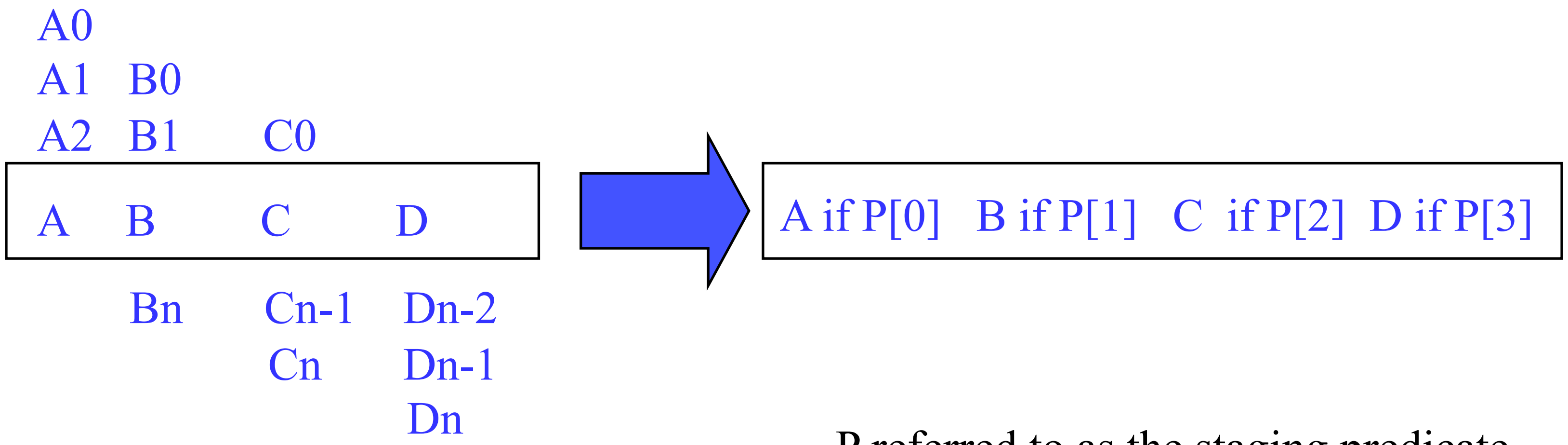
Peel off II-1 iterations for the prolog. Complete II-1 iterations in epilog

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



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