

EECS 583 – Class 13

Software Pipelining

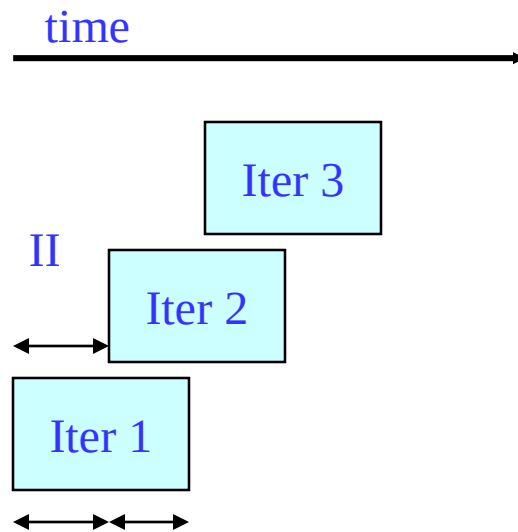
University of Michigan

October 13, 2021

Announcements + Reading Material

- ❖ Project discussion meetings – next week (Wed/Thu/Fri) + Monday Oct 25 (Wed Oct 27 is next regular class)
 - » Each group meets 10 mins with Yunjie/Ze and I
 - » Action item
 - Need to identify group members
 - Use piazza to recruit additional group members or express your availability
 - Think about project areas that you want to work on
 - » Google calendar signup available – see piazza
- ❖ Project proposals
 - » Due Wednesday, Oct 27, 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 & Yunjie & Ze, cc all your group members
- ❖ Today's class reading
 - » “Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops”, B. Rau, MICRO-27, 1994, pp. 63-74.
 - » “Code Generation Schema for Modulo Scheduled Loops”, B. Rau, M. Schlansker, and P. Tirumalai, MICRO-25, Dec. 1992.

Software Pipelining 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

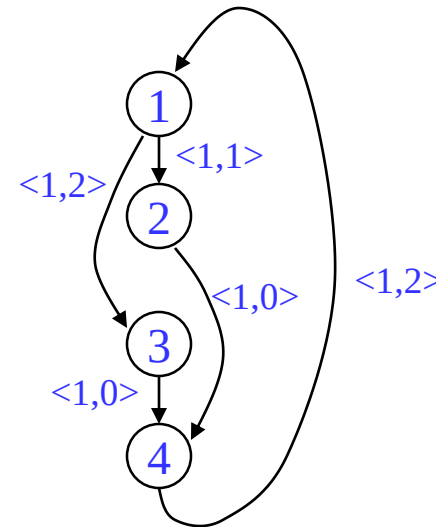
- » No resource is used at 2 points in time that are separated by an interval which is a multiple of II
- » I.E., within a single iteration, the same resource is never used more than 1x at the same time modulo II
- » Known as modulo constraint, where the name modulo scheduling comes from
- » Modulo reservation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

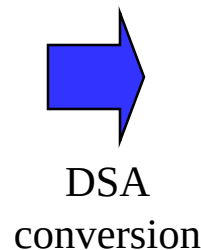
Impossible to overlap iterations because each iteration writes to the same register. So, we'll have to remove the anti and output dependences.

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 $\rightarrow r[n]$ 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
```



```
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
```

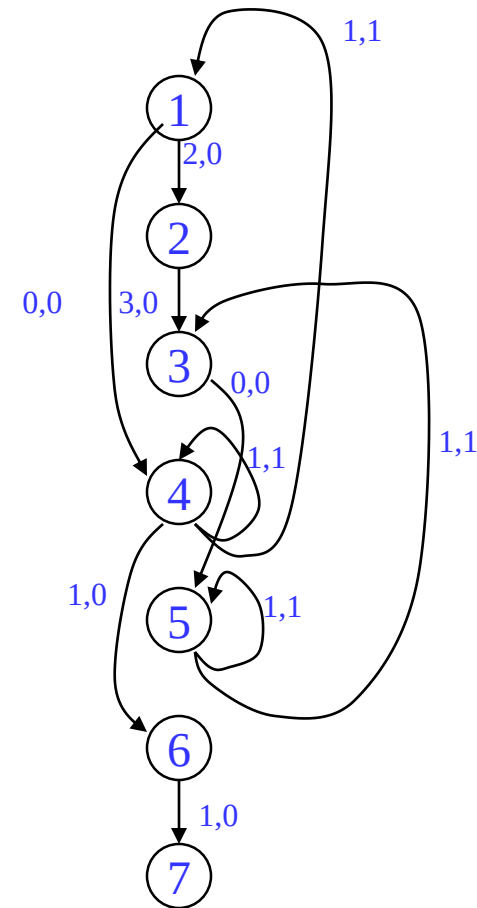
Physical Realization of EVRs

- ❖ 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

①

②

③

④

⑤

⑥

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 count(r) specifying the number of identical copies

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

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
```

$\text{ResMII} = \text{MAX} \quad (\text{uses}(r) / \text{count}(r))$

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

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

$\text{ResMII} = \text{MAX}(2, 2, 1) = 2$

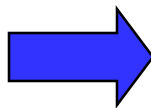
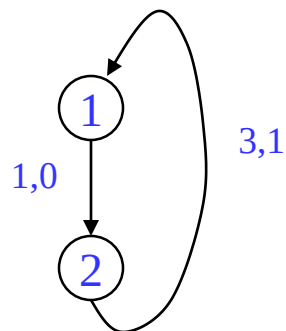
RecMII

Approach: Enumerate all irredundant elementary circuits in the dependence graph

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

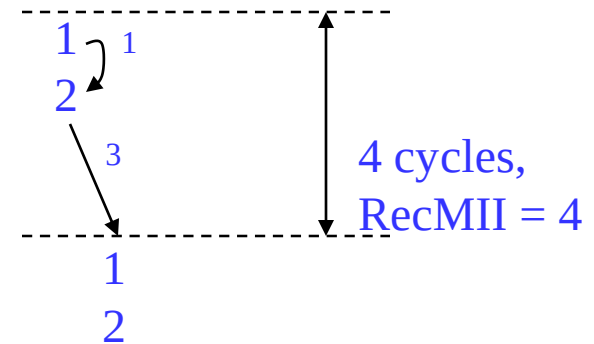
$\text{delay}(c)$ = total latency in dependence cycle c (sum of delays)

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



cycle

k
k+1
k+2
k+3
k+4
k+5



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

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

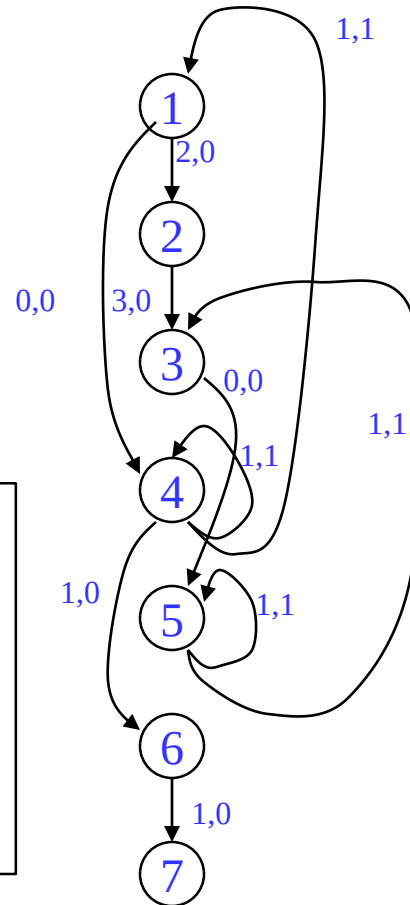
$$\text{RecMII} = 4/1 = 4$$

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
```

$\text{RecMII} = \text{MAX}(\text{delay}(c) / \text{distance}(c))$

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



<delay, distance>

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

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

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

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

$\text{RecMII} = \text{MAX}(1,1,1,1) = 1$

Then,

$\text{MII} = \text{MAX}(\text{ResMII}, \text{RecMII})$

$\text{MII} = \text{MAX}(2,1) = 2$

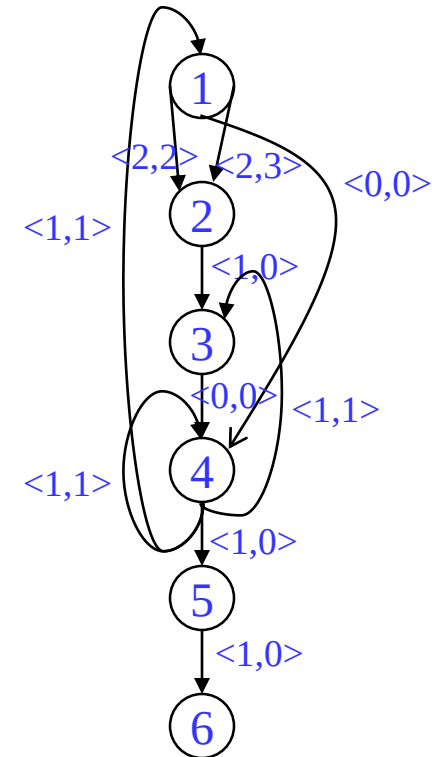
Class Problem

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:

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

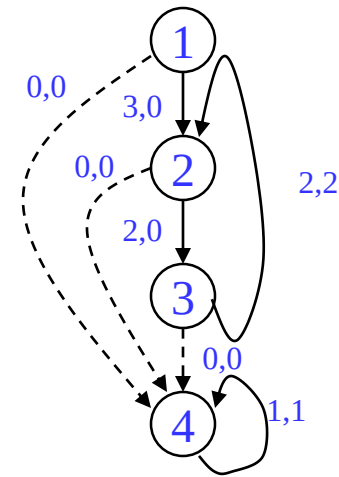
Cyclic:

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

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

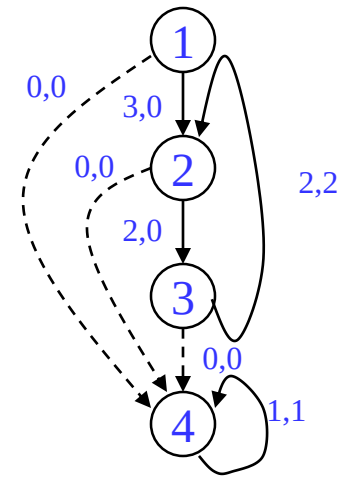
Calculating Height

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) =



Calculating Height Solution

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) = $H(4) + (1 - II*1)$ (Assume H(4) is 0 since not calculated yet)
 $0 + 1 - 2 = -1 \rightarrow 0$ (Always MAX answer with 0)
4. HeightR(3) = $\text{MAX}(H(4) + 0 - II*0 = 0 + 0 - 2*0 = 0,$
 $H(2) + 2 - II*2 = 0 + 2 - 2*2 = -2)$
 Assume H(2) is 0 since not calculated yet
 $= 0$
5. HeightR(2) = $\text{MAX}(H(4) + 0 - II*0 = 0 + 0 - 2*0 = 0,$
 $H(3) + 2 - II*0 = 0 + 2 - 2*0 = 2)$
 $= 2$
6. HeightR(1) = $\text{MAX}(H(4) + 0 - II*0 = 0 + 0 - 2*0 = 0,$
 $H(2) + 3 - II*0 = 2 + 3 - 2*0 = 5)$
 $= 5$
7. Now recalculate the heights to see if anything changes since HeightR(3) assumed wrong value for node 2
 HeightR(3) = $\text{MAX}(H(4) + 0 - II*0 = 0 + 0 - 2*0 = 0,$
 $H(2) + 2 - II*2 = 2 + 2 - 2*2 = 0)$
 $= 0 \rightarrow$ Unchanged, so no need to compute any other heights again



The Scheduling Window

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

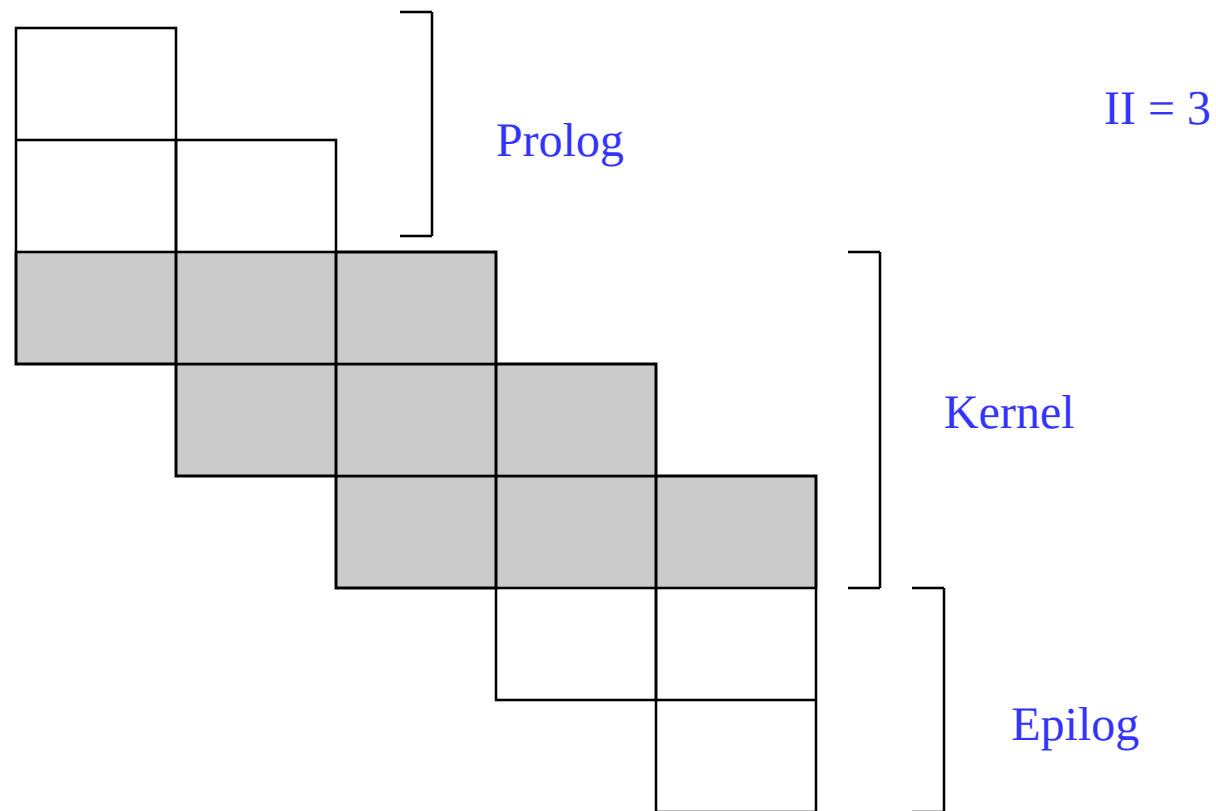
$$E(Y) = \underset{\text{for all } X = \text{pred}(Y)}{\text{MAX}} \begin{cases} 0, & \text{if } X \text{ is not scheduled} \\ \text{MAX } (0, \text{SchedTime}(X) + \text{EffDelay}(X, Y)), & \text{otherwise} \end{cases}$$

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.

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

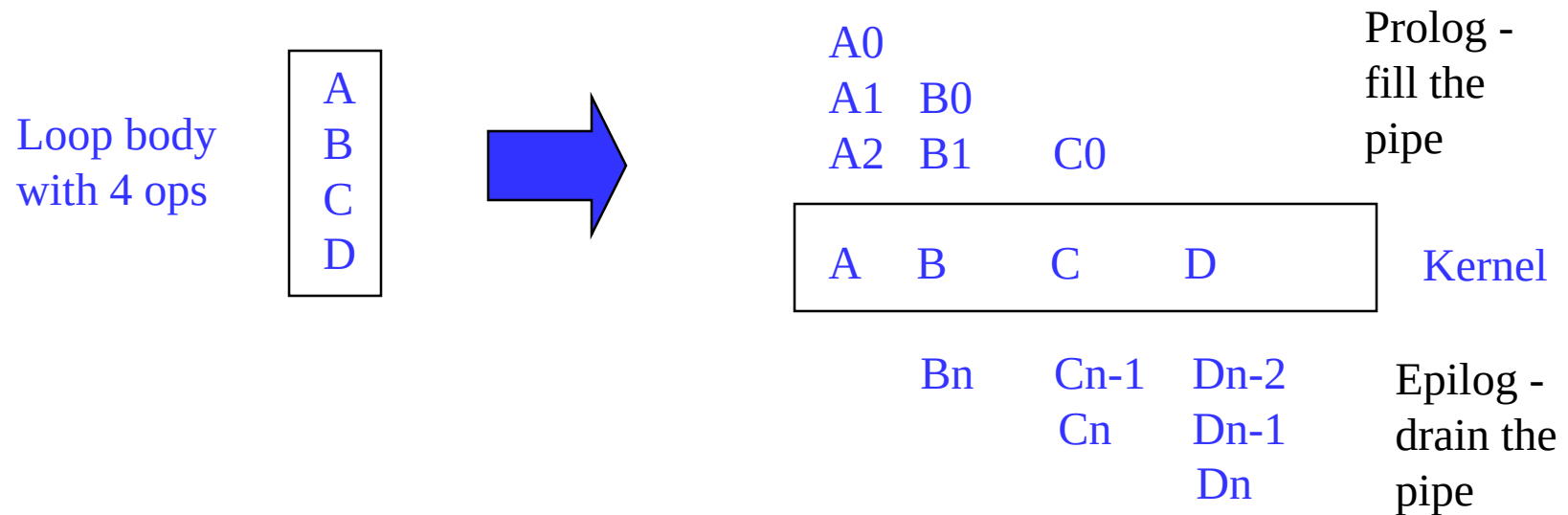
Loop Prolog and Epilog



Only the kernel involves executing full width of operations

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

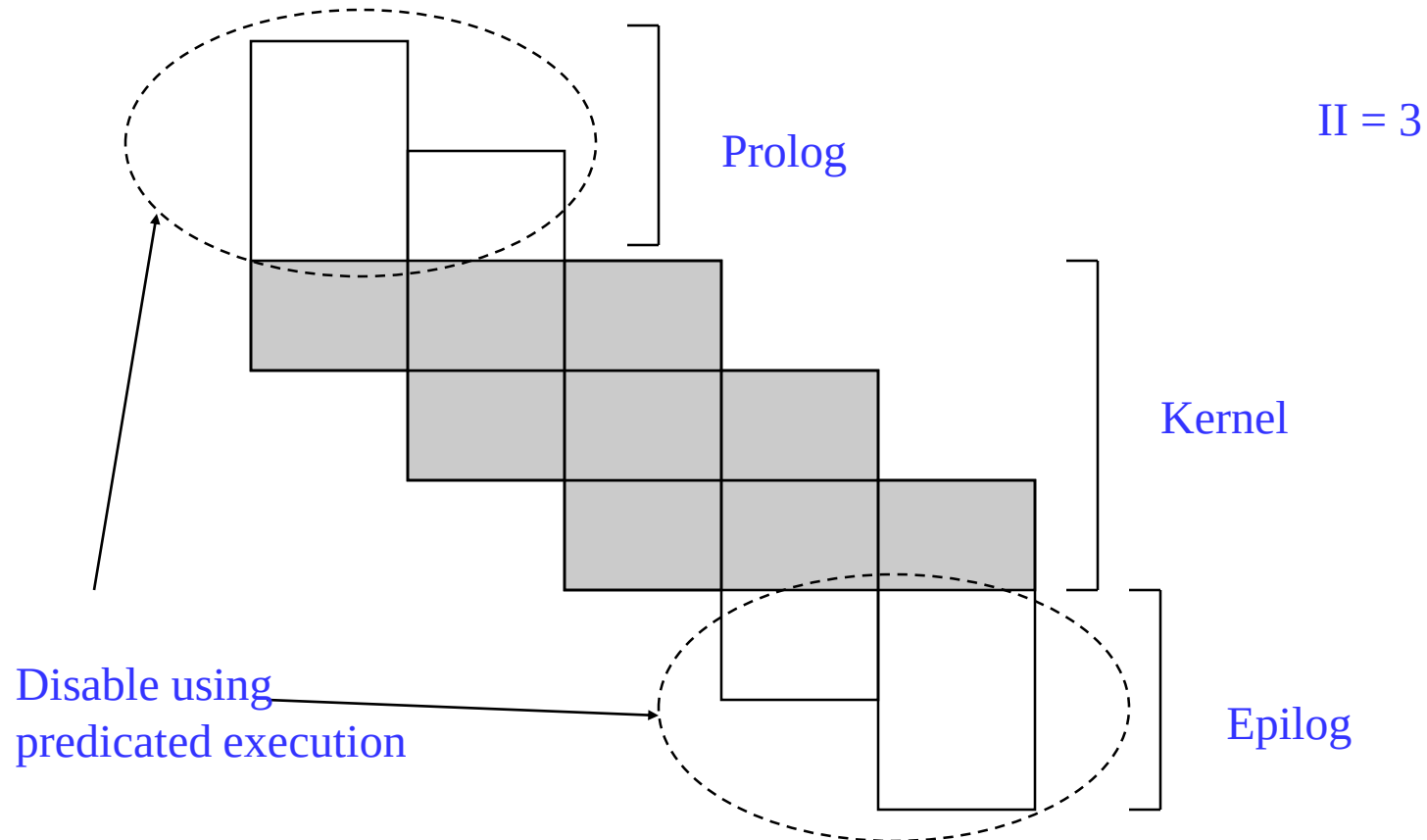
Separate Code for Prolog and Epilog



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

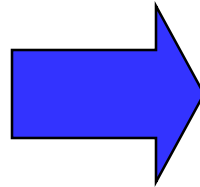
Kernel-only Code Using Rotating Predicates

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]
-----------	-----------	-----------	-----------

B_n

C_{n-1}

D_{n-2}

C_n

D_{n-1}

D_n

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
 - » SWP-BR (BRF)
 - 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

Homework

- ❖ Go through the following example
- ❖ We'll go through it in class after Fall Break and project meetings

Modulo Scheduling Example

resources: 4 issue, 2 alu, 1 mem, 1 br

latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

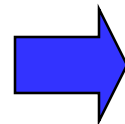
```
for (j=0; j<100; j++)  
    b[j] = a[j] * 26
```

Step1: Compute to loop into
form that uses LC

LC = 99

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
```



Loop:

```
1: r3 = load(r1)  
2: r4 = r3 * 26  
3: store (r2, r4)  
4: r1 = r1 + 4  
5: r2 = r2 + 4  
7: brlc Loop
```

Example – Step 2

resources: 4 issue, 2 alu, 1 mem, 1 br

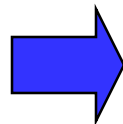
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

Step 2: DSA convert

LC = 99

Loop:

```
1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
7: brlc Loop
```



LC = 99

Loop:

```
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
  remap r1, r2, r3, r4
7: brlc Loop
```

Example – Step 3

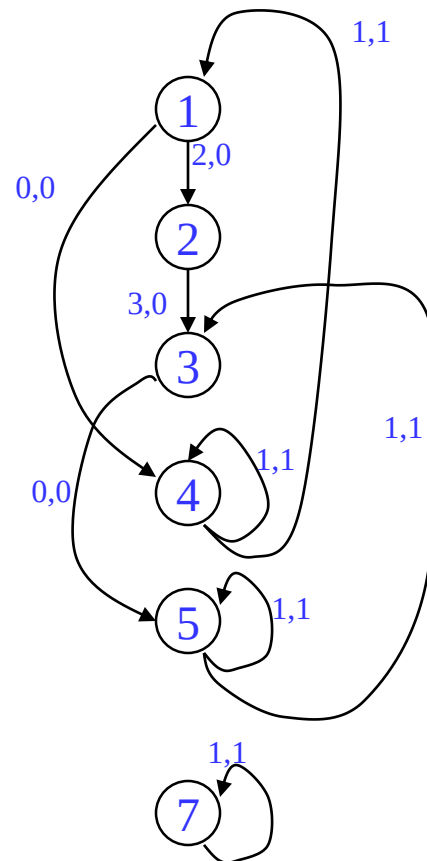
resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

Step3: Draw dependence graph
Calculate MII

LC = 99

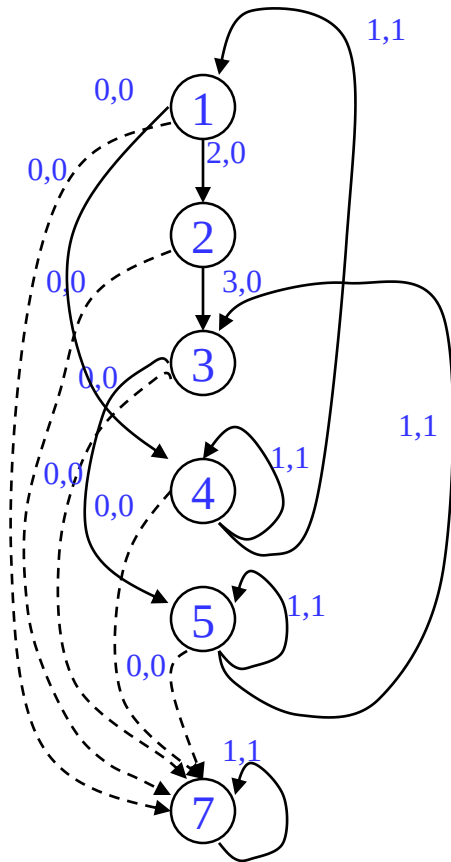
Loop:

```
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
  remap r1, r2, r3, r4
7: brlc Loop
```



RecMII = 1
RESMII = 2
MII = 2

Example – Step 4



Step 4 – Calculate priorities (MAX height to pseudo stop node)

Iter1

1: H = 5

2: H = 3

3: H = 0

4: H = 0

5: H = 0

7: H = 0

Iter2

1: H = 5

2: H = 3

3: H = 0

4: H = 4

5: H = 0

7: H = 0

Example – Step 5

resources: 4 issue, 2 alu, 1 mem, 1 br

latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

Schedule brlc at time II - 1

LC = 99

Loop:

```
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
remap r1, r2, r3, r4
7: brlc Loop
```

Rolled
Schedule

0	
1	7

Unrolled
Schedule

0	
1	
2	
3	
4	
5	
6	

	alu0	alu1	mem	br
0				
1				X

MRT

Example – Step 6

Step6: Schedule the highest priority op

Op1: E = 0, L = 1

Place at time 0 (0 % 2)

LC = 99

Loop:

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
remap r1, r2, r3, r4
7: brlc Loop

Rolled
Schedule

0	1
1	7

Unrolled
Schedule

0	1
1	
2	
3	
4	
5	
6	

	alu0	alu1	mem	br
0			X	
1				X

MRT

Example – Step 7

Step7: Schedule the highest priority op

Op4: E = 0, L = 1

Place at time 0 (0 % 2)

LC = 99

Loop:

```
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
remap r1, r2, r3, r4
7: brlc Loop
```

Rolled
Schedule

0	1	4
1	7	

Unrolled
Schedule

0	1	4
1		
2		
3		
4		
5		
6		

	alu0	alu1	mem	br
0	X		X	
1				X

MRT

Example – Step 8

Step8: Schedule the highest priority op

Op2: E = 2, L = 3

Place at time 2 (2 % 2)

LC = 99

Loop:

```
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
remap r1, r2, r3, r4
7: brlc Loop
```

Rolled
Schedule

0	1	4	2
1	7		

Unrolled
Schedule

0	1	4
1		
2	2	
3		
4		
5		
6		

	alu0	alu1	mem	br
0	X	X	X	
1				X

MRT

Example – Step 9

Step9: Schedule the highest priority op

Op3: E = 5, L = 6

Place at time 5 (5 % 2)

LC = 99

Loop:

```

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
remap r1, r2, r3, r4
7: brlc Loop
    
```

Rolled
Schedule

0	1	2	4
1	7	3	

Unrolled
Schedule

0	1	4
1		
2	2	
3		
4		
5	3	
6		

	alu0	alu1	mem	br
0	X	X	X	
1			X	X

MRT

Example – Step 10

Step10: Schedule the highest priority op

Op5: E = 5, L = 6

Place at time 5 (5 % 2)

LC = 99

Loop:

```
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
remap r1, r2, r3, r4
7: brlc Loop
```

Rolled
Schedule

0	1	2	4
1	7	3	5

Unrolled
Schedule

0	1	4
1		
2	2	
3		
4		
5	3	5
6		

	alu0	alu1	mem	br
0	X	X	X	
1	X		X	X

MRT

Example – Step 11

Step11: calculate ESC, $SC = \text{ceiling}(\text{max unrolled sched length} / ii)$
 unrolled sched time of branch = rolled sched time of br + $(ii * \text{esc})$

$SC = 6 / 2 = 3$, $ESC = SC - 1$
 time of br = $1 + 2 * 2 = 5$

LC = 99

Loop:

```

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
remap r1, r2, r3, r4
7: brlc Loop
    
```

Rolled
Schedule

0	1	2	4
1	7	3	5

Unrolled
Schedule

0	1	4	
1			
2	2		
3			
4			
5	3	5	7
6			

	alu0	alu1	mem	br
0	X	X	X	
1	X		X	X

MRT

Example – Step 12

Finishing touches - Sort ops, initialize ESC, insert BRF and staging predicate, initialize staging predicate outside loop

LC = 99
ESC = 2
p1[0] = 1

Loop:

1: r3[-1] = load(r1[0]) if p1[0]
2: r4[-1] = r3[-1] * 26 if p1[1]
4: r1[-1] = r1[0] + 4 if p1[0]
3: store (r2[0], r4[-1]) if p1[2]
5: r2[-1] = r2[0] + 4 if p1[2]
7: brlc Loop if p1[2]

Staging predicate, each
successive stage increment
the index of the staging predicate
by 1, stage 1 gets px[0]

Unrolled
Schedule

0	1	4	Stage 1
1			
2	2		Stage 2
3			
4			Stage 3
5	3	5	
6			

Example – Dynamic Execution of the Code

LC = 99
ESC = 2
p1[0] = 1

Loop: 1: r3[-1] = load(r1[0]) if p1[0]
2: r4[-1] = r3[-1] * 26 if p1[1]
4: r1[-1] = r1[0] + 4 if p1[0]
3: store (r2[0], r4[-1]) if p1[2]
5: r2[-1] = r2[0] + 4 if p1[2]
7: brlc Loop if p1[2]

Total time = $II(\text{num_iteration} + \text{num_stages} - 1)$
 $= 2(100 + 3 - 1) = 204 \text{ cycles}$

time: ops executed

0: 1, 4

1:

2: 1,2,4

3:

4: 1,2,4

5: 3,5,7

6: 1,2,4

7: 3,5,7

...

198: 1,2,4

199: 3,5,7

200: 2

201: 3,5,7

202: -

203 3,5,7