EECS 583 – Class 14
Modulo Scheduling Reloaded

University of Michigan

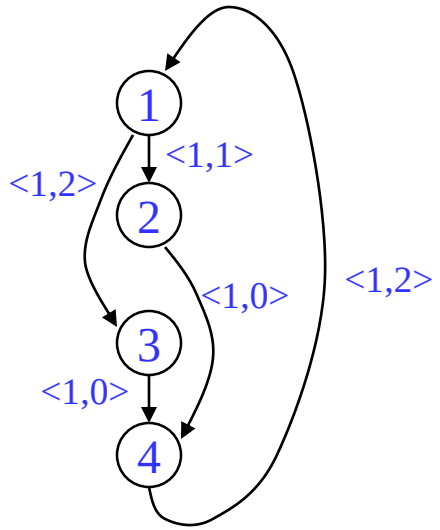
October 27, 2021

Announcements + Reading Material

- Project proposals
  - Due tonight, 11:59pm
  - 1 paragraph summary of what you plan to work on, 1-2 references
  - Email to me & Yunjie & Ze, cc all your group members
- Exam – Next Wednes, Nov 3 10:30-12:30
  - Exam review – next Monday
  - Exam scope: Covers all lecture material through today's class
  - Exam format: Hybrid (Virtual or in-person) – Would anyone want in-person?
    - In-person: 10:30-12:15 + 15 mins extra time, walk outside to get questions answered
    - Virtual: 10:30-12:15 + 30 mins extra time (15 mins extra for printing, scanning, uploading), post private questions on piazza to get answers
    - Questions answered up to 12:30
- Today's class reading
  - "Code Generation Schema for Modulo Scheduled Loops", B. Rau, M. Schlansker, and P. Tirumalai, MICRO-25, Dec. 1992.
- Next class reading - None

From Last Time: 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>

From Last Time: Dynamic Single Assignment (DSA) Form

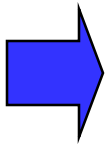
Impossible to overlap iterations because each iteration writes to the same register. So, we'll have to remove the anti and output dependences.

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
```

From Last Time: ResMII Example

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

ResMII = MAX (uses(r) / count(r))

uses(r) = number of times the resource is used in 1 iteration

```
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

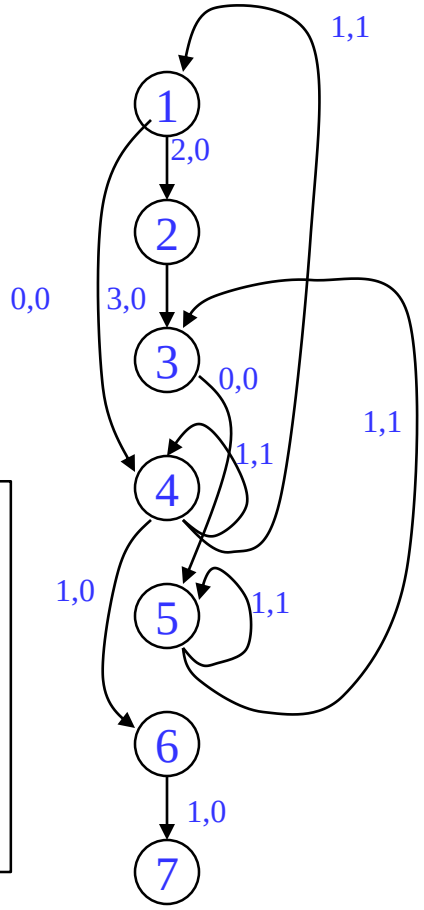
From Last Time: RecMII Example

```
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
```

RecMII = MAX(delay(c) / 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, 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

From Last Time: Priority Function

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

Acyclic:

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

Cyclic:

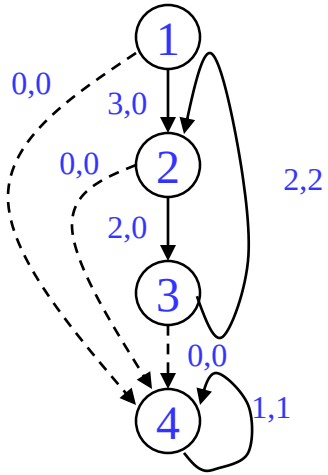
$$HeightR(X) = \begin{cases} 0, \text{ if X has no successors} \\ \max_{\text{for all Y = succ(X)}} ((HeightR(Y) + EffDelay(X,Y)), \text{ otherwise} \end{cases}$$

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

From Last Time: 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) = H(4) + (1 – II*1) (Assume H(4) is 0 since not calculated yet
   0 + 1-2 = -1 → 0 (Always MAX answer with 0)

4. HeightR(3) = 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) = 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) = 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) = MAX(H(4) + 0 – II*0 = 0 + 0 – 2*0 = 0,
   H(2) + 2 – II*2 = 2 + 2 – 2*2 = 0)
   = 0 → 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:

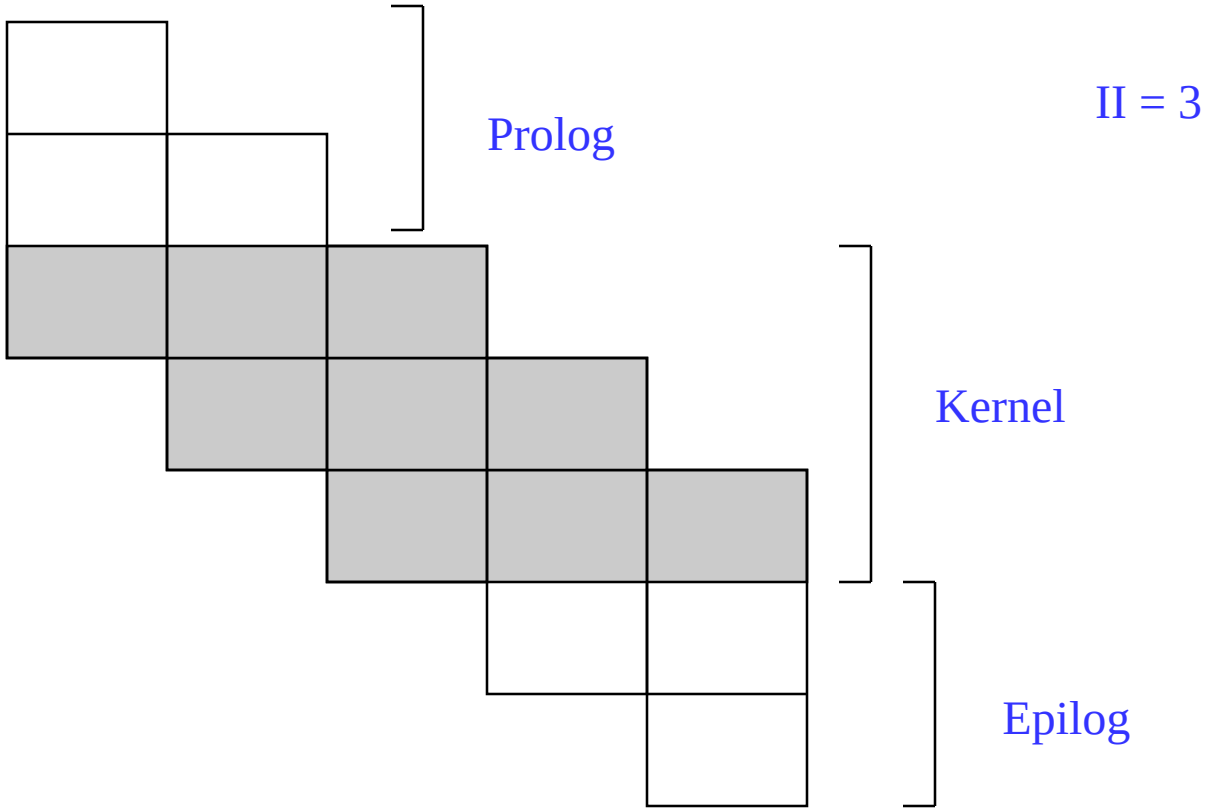
$$E(Y) = \max_{\text{for all } X = \text{pred}(Y)} \begin{cases} 0, \text{ if } X \text{ is not scheduled} \\ \max(0, \text{SchedTime}(X) + \text{EffDelay}(X,Y)), \text{ otherwise} \end{cases}$$

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.

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

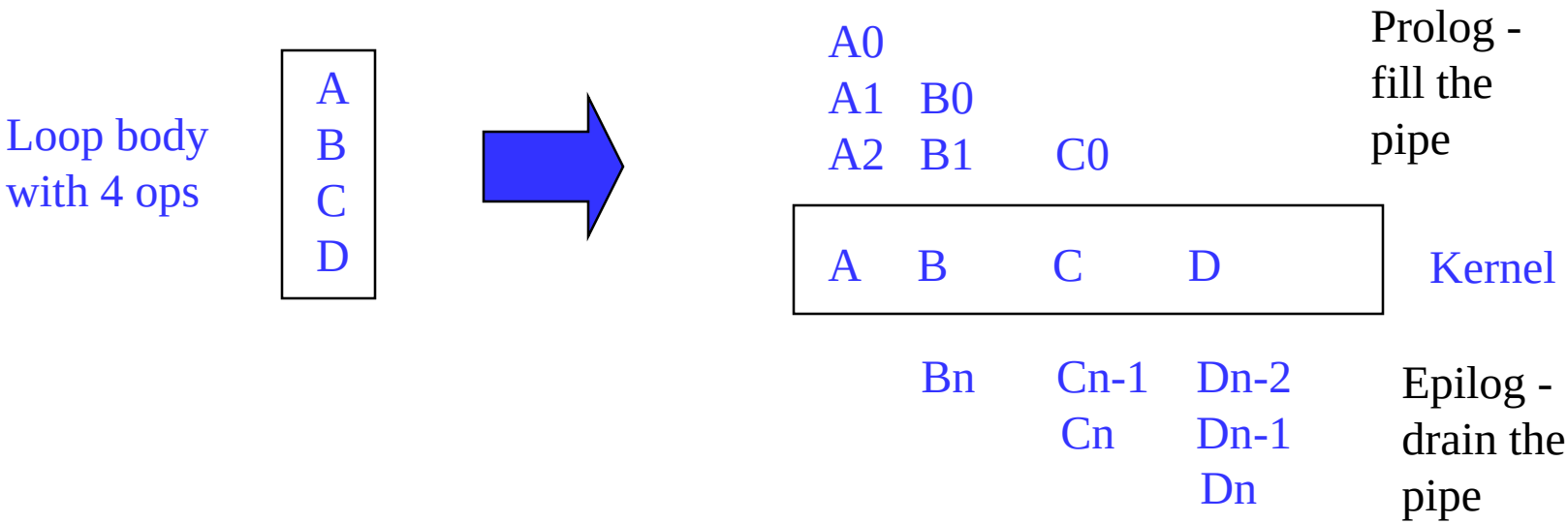
Loop Prolog and Epilog



Only the kernel involves executing full width of operations

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

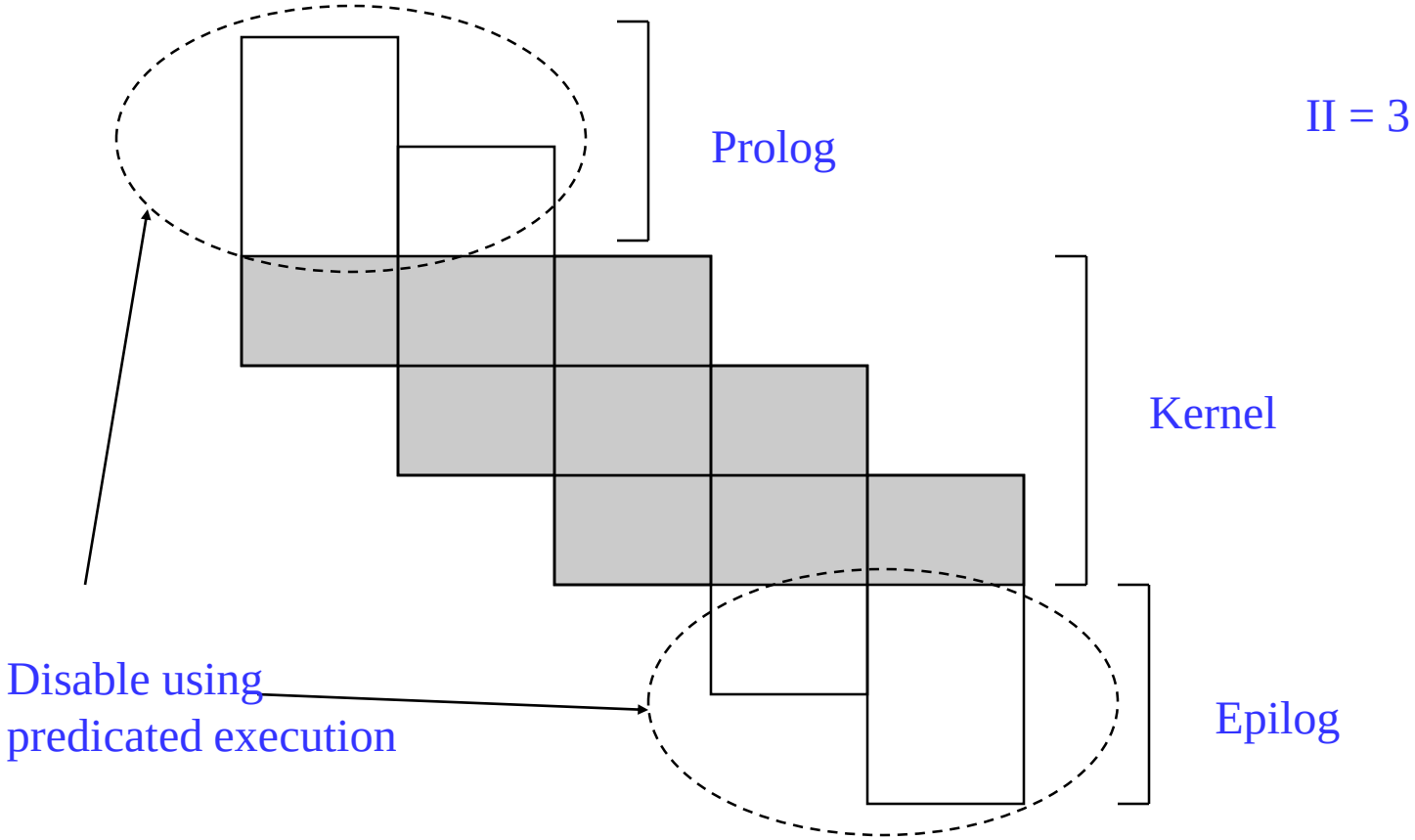
Separate Code for Prolog and Epilog



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

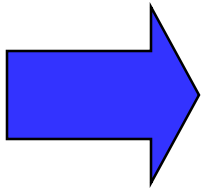


Execute loop kernel on every iteration, but for prolog and epilog selectively disable the appropriate operations to fill/drain the pipeline

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
  - SWP-BR (BRLC)
    - 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.B.B.F;
```

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

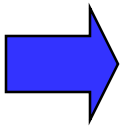
Modulo Scheduling Example

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

```
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
```



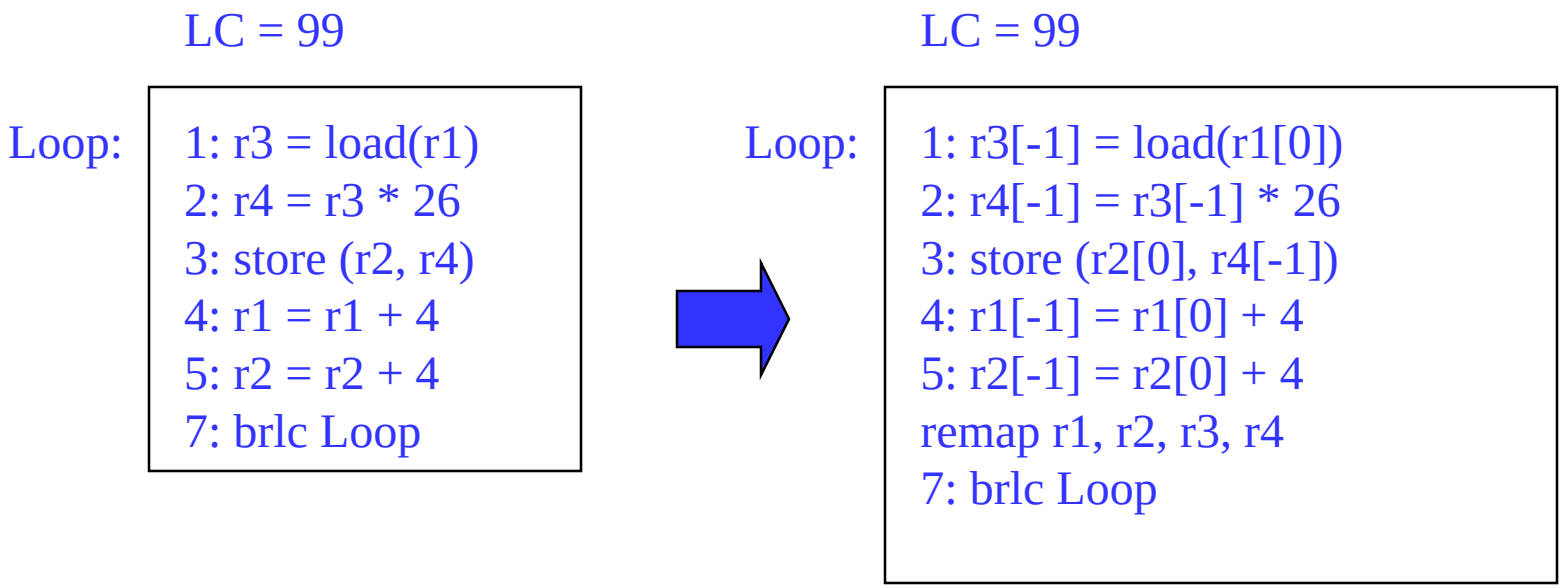
LC = 99

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

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



Example – Step 3

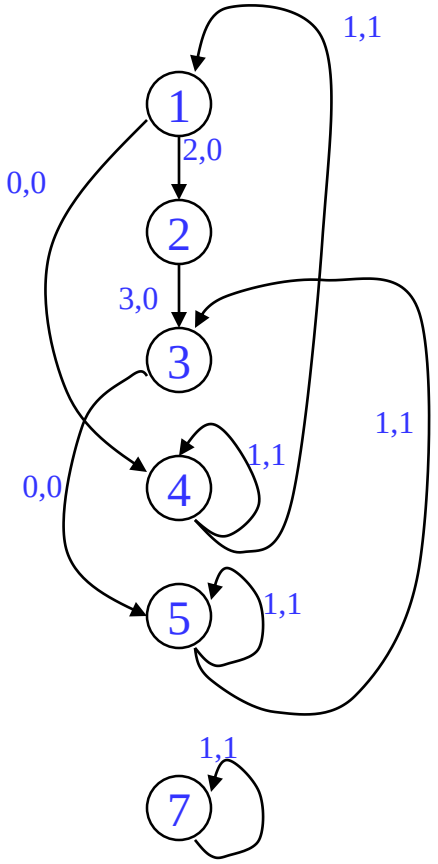
resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

Step3: Draw dependence graph
Calculate MII

LC = 99

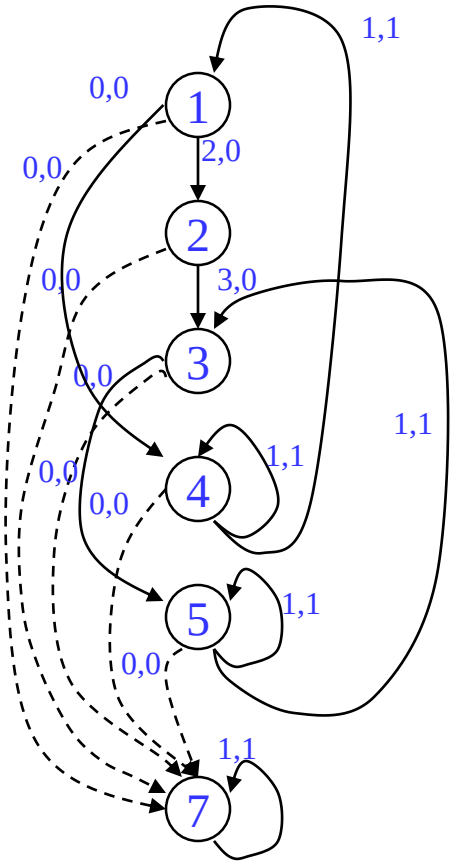
Loop:

```
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	Iter2
1: H = 5	1: H = 5
2: H = 3	2: H = 3
3: H = 0	3: H = 0
4: H = 0	4: H = 4
5: H = 0	5: H = 0
7: 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	

MRT

	alu0	alu1	mem	br
0				
1				X

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	

MRT

	alu0	alu1	mem	br
0			X	
1				X

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	

MRT

	alu0	alu1	mem	br
0	X		X	
1				X

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	

MRT

	alu0	alu1	mem	br
0	X	X	X	
1				X

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	

MRT

	alu0	alu1	mem	br
0	X	X	X	
1			X	X

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	

MRT

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

Example – Step 11

Step11: calculate ESC, SC = ceiling(max unrolled sched length / ii)
unrolled sched time of branch = rolled sched time of br + (ii*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	

MRT

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

Example – Step 12

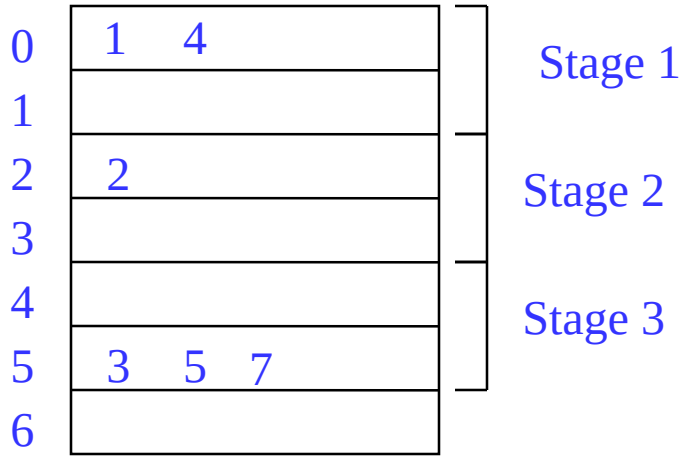
Finishing touches - Sort ops, initialize ESC, insert BRF and staging predicate, initialize staging predicate outside loop

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

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

```
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]
```

Unrolled Schedule



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(num_iteration + num_stages – 1)
= 2(100 + 3 – 1) = 204 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

Homework Problem

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

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

```
      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
```

How many resources of each type are required to achieve an II=1 schedule?

If the resources are non-pipelined, how many resources of each type are required to achieve II=1

Assuming pipelined resources, generate the II=1 modulo schedule.

Homework Problem – Answers in Red

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

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

```
      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
```

How many resources of each type are required to achieve an II=1 schedule?
For II=1, each operation needs a dedicated resource, so: 3 ALU, 2 MEM, 1 BR

If the resources are non-pipelined, how many resources of each type are required to achieve II=1
Instead of 1 ALU to do the multiplies, 3 are needed, and instead of 1 MEM to do the loads, 2 are needed.
Hence: 5 ALU, 3 MEM, 1 BR

Assuming pipelined resources, generate the II=1 modulo schedule.
See next few slides

Problem continued

Assume II=1 so resources are: 3 ALU, 2 MEM, 1 BR

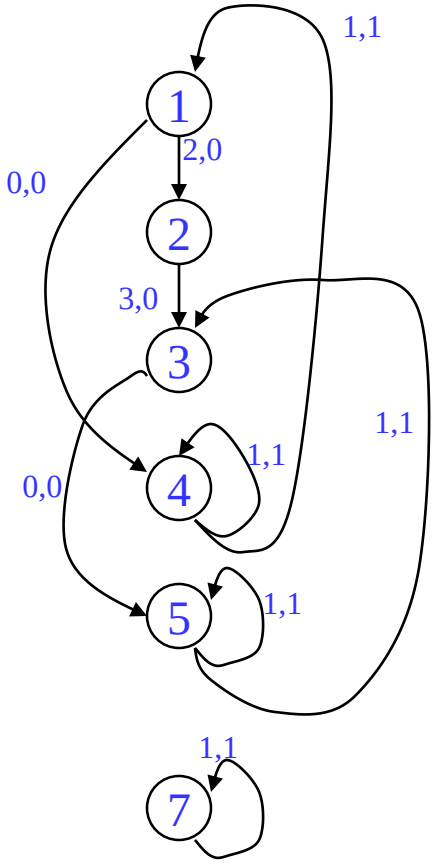
DSA converted code below (same as example in class)

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
```

Dependence graph (same as example in class)



RecMII = 1
RESMII = 1
MII = MAX(1,1) = 1

Priorities
1: H = 5
2: H = 3
3: H = 0
4: H = 4
5: H = 0
7: H = 0

Problem continued

resources: 3 alu, 2 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 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	7 1 4 2 3 5

Unrolled Schedule

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

Scheduling steps:

Schedule brlc at time II-1
Schedule op1 at time 0
Schedule op4 at time 0
Schedule op2 at time 2
Schedule op3 at time 5
Schedule op5 at time 5
Schedule op7 at time 5

MRT

	alu0	alu1	alu2	m1	m2	br
0	X	X	X	X	X	X

Problem continued

The final loop consists of a single MultiOp containing 6 operations, each predicated on the appropriate staging predicate. Note register allocation still needs to be performed.

```
    LC = 99
Loop:
    r3[-1] = load(r1[0]) if p1[0]; r4[-1] = r3[-1] * 26 if p1[2]; store (r2[0], r4[-1]) if p1[5]; r1[-1] = r1[0] + 4 if p1[0]; r2[-1] = r2[0] + 4 if p1[5]; brlc Loop
```

What if We Don't Have Hardware Support for Modulo Scheduling?

- No predicates
  - Predicates enable kernel-only code by selectively enabling/disabling operations to create prolog/epilog
  - Now must create explicit prolog/epilog code segments
- No rotating registers
  - Register names not automatically changed each iteration
  - Must unroll the body of the software pipeline, explicitly rename
    - Consider each register lifetime i in the loop
    - Kmin = min unroll factor = MAXi (ceiling((Endi – Starti) / II))
    - Create Kmin static names to handle maximum register lifetime
  - Apply modulo variable expansion