Loop Scheduling

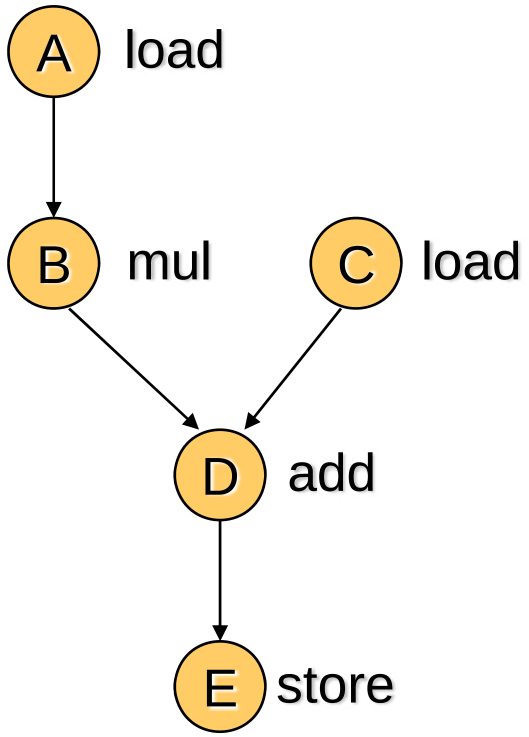
Compilers for High Performance Architectures

Loop Scheduling

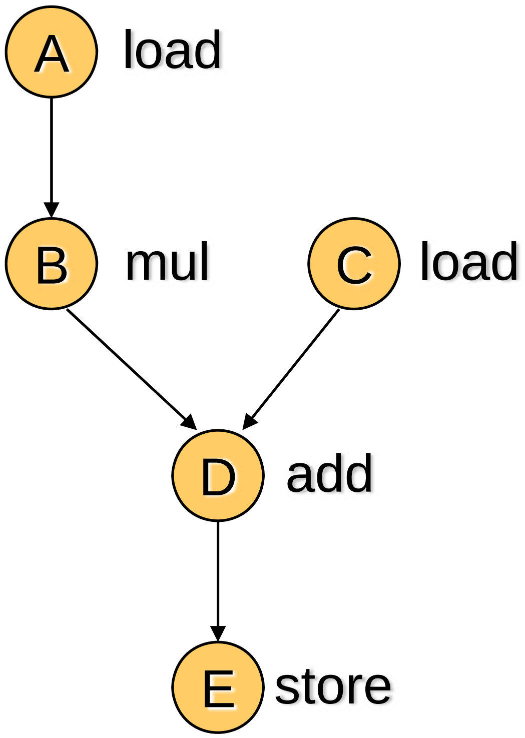
- Dependence Graph

```
do i=0,N
  y(i)=y(i)+a*x(i)
endo

do i=0,N
  A: t1 = x(i)   ; load
  B: t2 = a*t1   ; mul
  C: t3 = y(i)   ; load
  D: t4 = t2 + t3; add
  E: y(i) = t4   ; store
enddo
```


$$\begin{tikzcd}
A \text{ load} \arrow[d] & \\
B \text{ mul} \arrow[dr] & C \text{ load} \arrow[d] \\
& D \text{ add} \arrow[d] \\
& E \text{ store}
\end{tikzcd}$$

Loop Scheduling



Cycle	Load 1	Load 2	add	mul	store
0	**A**	**C**			
1					
2			**B**		
3					
4				**D**	
5					
6					**E**

5 pipelined units

- 2 load units 2 cycles
- 1add unit 2 cycles
- 1 mul unit 2 cycles
- 1 store unit 1 cycle

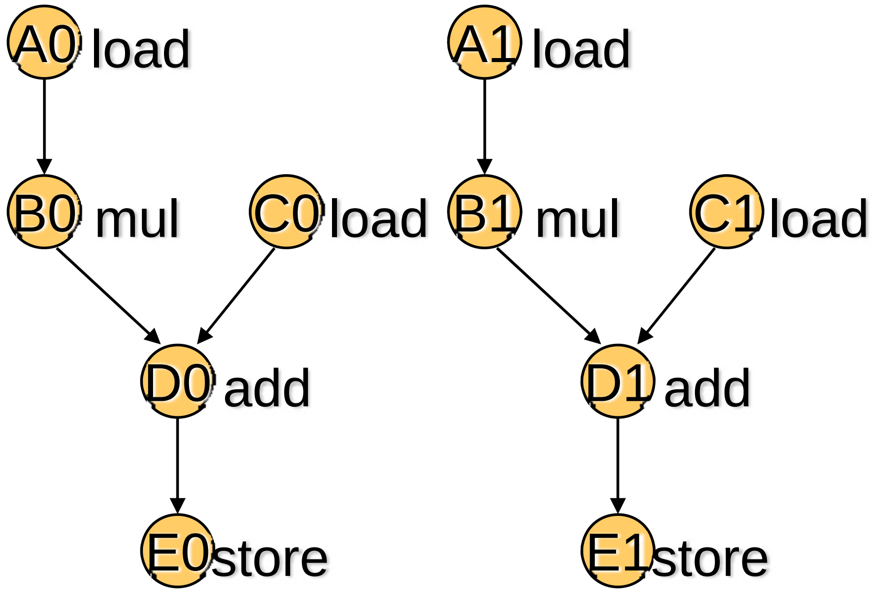
7 cycles per iteration

Maximum ILP = 5 Inst/cycle

Achieved ILP = 0.71 Inst/cycle

Schedule the unrolled loop

```
do i=0,N
      y(i+0)=y(i+0)+a*x(i+0)
      y(i+1)=y(i+1)+a*x(i+1)
endo
```



Cycle	Load 1	Load 2	add	mul	store
0	**A0**	**C0**			
1	**A1**	**C1**			
2			**B0**		
3			**B1**		
4				**D0**	
5				**D1**	
6					**E0**
7					**E1**

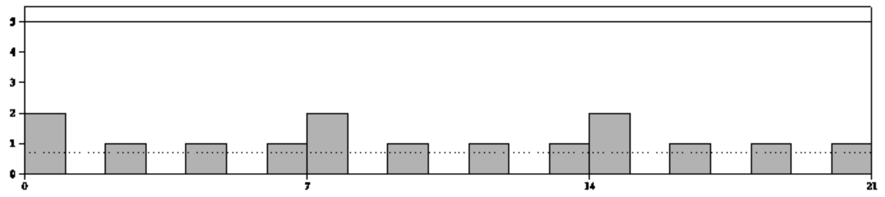
8 cycles per 2 iterations

Maximum ILP = 5 Inst/cycle

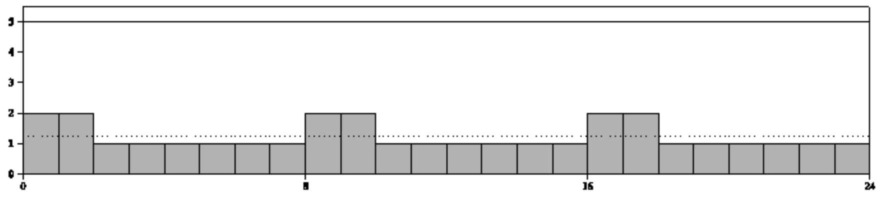
Achieved ILP = 1.25 Inst/cycle

ILP versus unroll degree

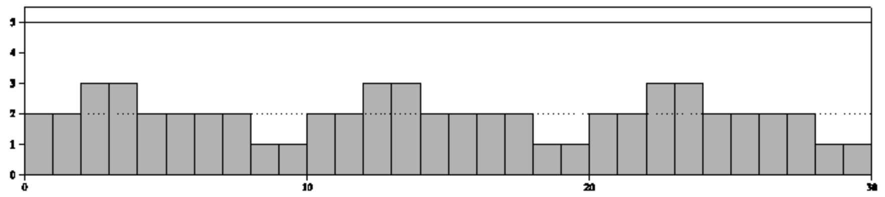
No unroll



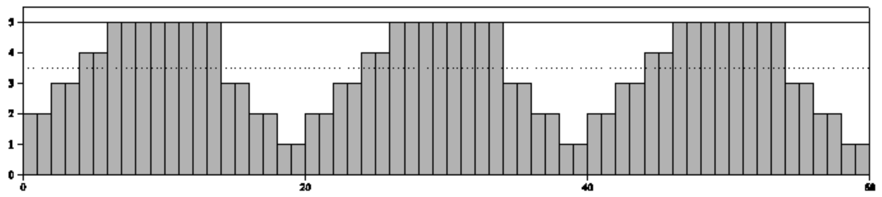
Unroll = 2



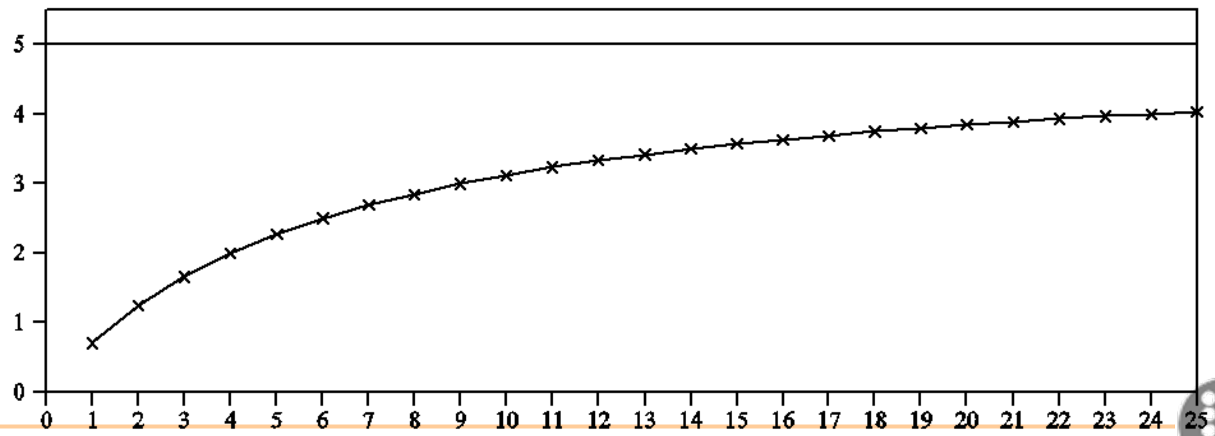
Unroll = 4



Unroll = 14



ILP versus
Unroll degree



ILP vs unroll degree

- To achieve maximum parallelism
- Fully unroll the loop
- We have found a repeating patern !!!

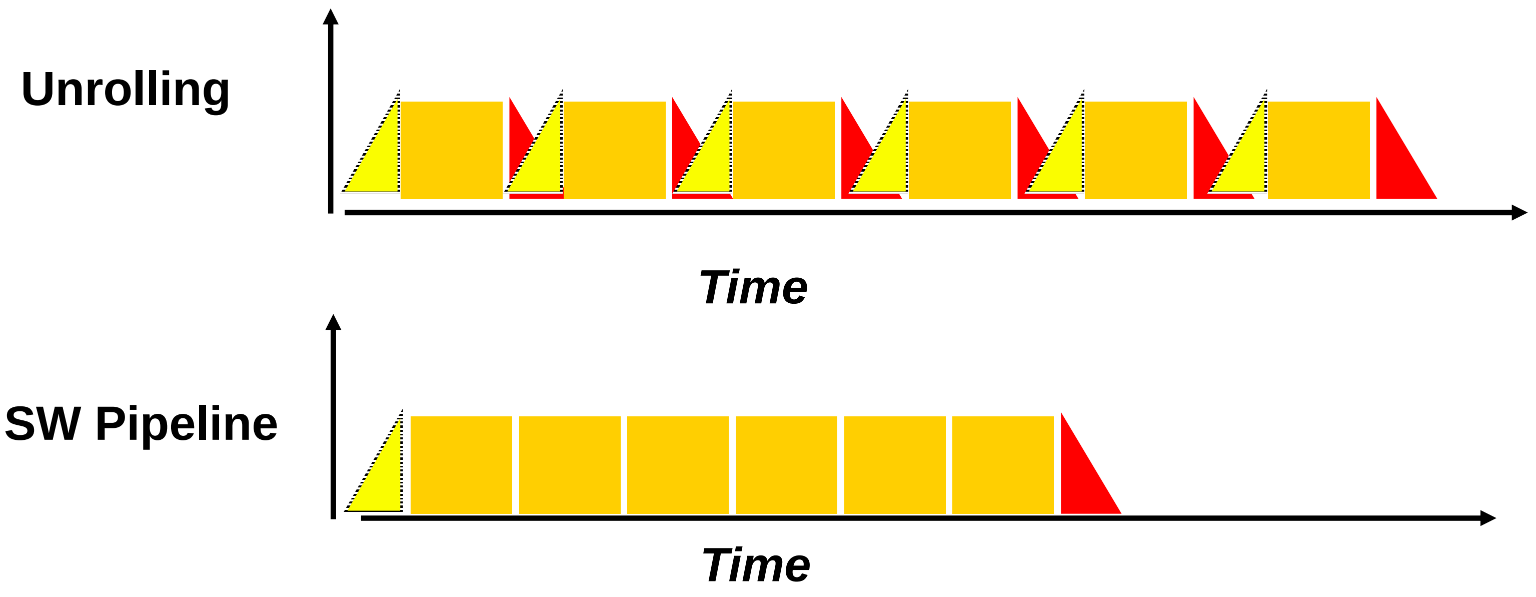
Cycle	Load 1	Load 2	add	mul	store
0	A0	C0			
1	A1	C1			
2	A2	C2	B0		
3	A3	C3	B1		
4	A4	C4	B2	D0	
5	A5	C5	B3	D1	
6	A6	C6	B4	D2	E0
7	A7	C7	B5	D3	E1

i-1	Ai-1	Ci-1	Bi-3	Di-5	Ei-7
i	Ai	Ci	Bi-2	Di-5	Ei-6

n	An	Cn	Bn-2	Dn-5	En-7
n+1			Bn-1	Dn-4	En-6
n+2			Bn	Dn-3	En-5
n+3				Dn-2	En-4
N+4				Dn-1	En-3
N+5				Dn	En-2
N+6					En-1
n+7					En

Software Pipelining

- What is Software Pipelining?
  - A technique for scheduling instructions to exploit ILP in inner loops
    - Many programs spend most of their time in inner loops (e.g. 80% in Perfect Club)
    - We can exploit parallelism between loop iterations
    - Achieves same effect as fully unrolling the loop
- More efficient than unrolling (but also more limited)



Software pipelining

- A repeating pattern can be detected or constructed
  - kernel
- By iterating over the kernel we can achieve the same effect as fully unrolling the loop
- We need code before/after the loop to fill/drain the "pipe":
  - prologue
  - epilogue

Cycle	Load 1	Load 2	add	mul	store
0	**A0**	**C0**			
1	**A1**	**C1**			
2	**A2**	**C2**	**B0**		
3	**A3**	**C3**	**B1**		
4	**A4**	**C4**	**B2**	**D0**	
5	**A5**	**C5**	**B3**	**D1**	

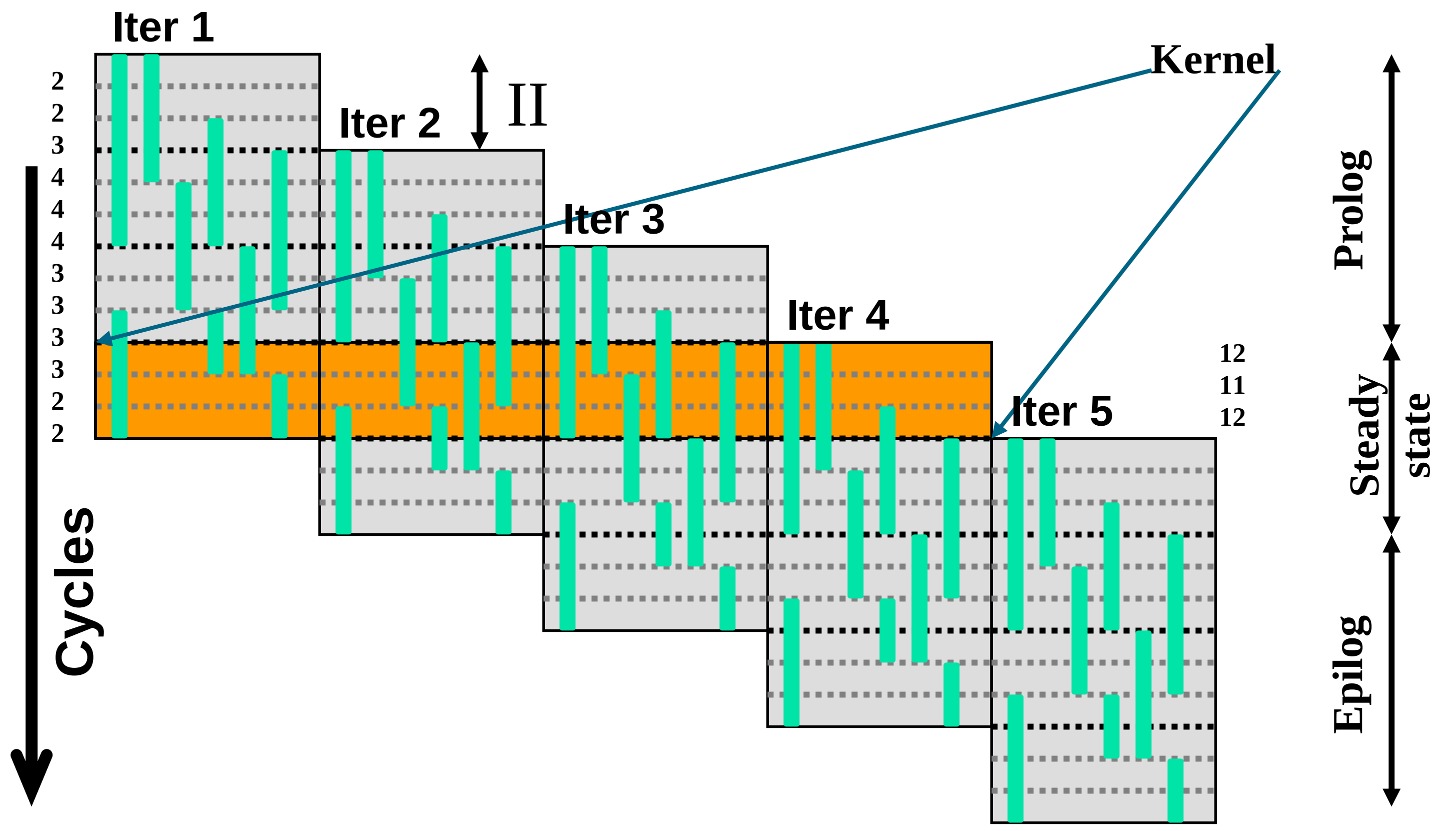
do i=6, N

i	**Ai**	**Ci**	**Bi-2**	**Di-5**	**Ei-6**

endo

n+1			**Bn-1**	**Dn-4**	**En-6**
n+2			**Bn**	**Dn-3**	**En-5**
n+3				**Dn-2**	**En-4**
N+4				**Dn-1**	**En-3**
N+5				**Dn**	**En-2**
N+6					**En-1**
n+7					**En**

Software pipelining



Software pipelining

- The schedule of an iteration can be divided in stages
- Every II (Initiation Interval) cycles a new iteration is started
- Every II cycles active iterations transition to the next stage
- Several iterations overlap their execution in different stages
- Very similar to the behavior of hardware pipelines hence the term “Software Pipelining”

Software Pipelining vs. Unrolling

- Sw pipelining optimizes the loop throughput
  - Assumption: latency of an iteration is not important
  - Unrolling optimizes the latency of n iterations

- Sw pipelining limits code expansion to prologue / epilogue
  - Even Prologue/Epilogue could be removed with additional hardware support (rotating registers and predicates)

- Sw pipelining is practically limited to loops without control flow
  - Control flow in the loop must be predicated out
  - This may be very inefficient for unbalanced paths

- Efficient sw pipelining requires hardware support
  - Rotating registers, predicates and special branches

Software pipelining

- Main Software pipelining objective:
  - Find an instruction schedule for the kernel with the smallest initiation interval so that throughput is maximized
- Intuitive approaches: Keep unrolling and scheduling the loop until a repeating pattern is found
  - Detecting patterns is difficult
  - Results may be suboptimal
  - Convergence to a valid schedule is not guaranteed
- Direct approach: Build directly the schedule
  - NP complexity
  - Optimal approaches
  - Heuristic approaches
    - Modulo scheduling: best known and practical approach

Modulo Scheduling

Compilers for High Performance Architectures

Class outline

- Modulo scheduling
- Added difficulties for modulo scheduling
- Iterative modulo scheduling
- Code generation schemas
  - Code size
- Hardware support for modulo scheduling
  - Rotating register files
  - Rotating predicates
  - Code Schemas with hardware support
- Register requirements of software pipelined loops

Modulo Scheduling

- A software pipelining framework
  - Produce a schedule for one iteration so that no data dependence and no resource conflict arises when the same schedule is repeated at regular intervals
  - The constant interval between the start of successive iterations is called initiation interval (II)
- Modulo scheduling steps
  - Choose a candidate initial II (MII or minimum initiation interval)
  - MII is a lower bound of the II
  - Starting from MII, try to find a legal schedule
  - Increases the II until we can find a legal schedule

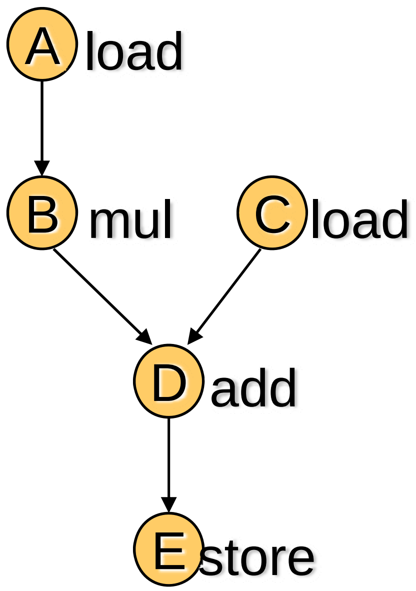
Modulo Scheduling Notes

- Scheduling a loop body within a MS framework is very different than scheduling acyclic regions
  - MS is not concerned with the schedule length
  - The II is what determines the loop performance: as long as a schedule satisfies II requirements, length is irrelevant
- To enforce constraints, MS uses a “modulo reservation table”
  - It checks for resource conflicts in the current cycle and in all schedule cycles that differ by II from the current cycle
- Modulo scheduling is near-optimal in 96-99% of simple loops

Initiation Interval

- Choosing a sensible MII improves the speed of the modulo scheduler
  - We can compute a good MII as the maximum of
    - A resource-constrained II (ResMII, derived from a mapping from the loop body to the target resources)
    - A recurrence-constrained II (RecMII, derived from the inter-iteration data dependence graph of the loop)

Modulo Scheduling example

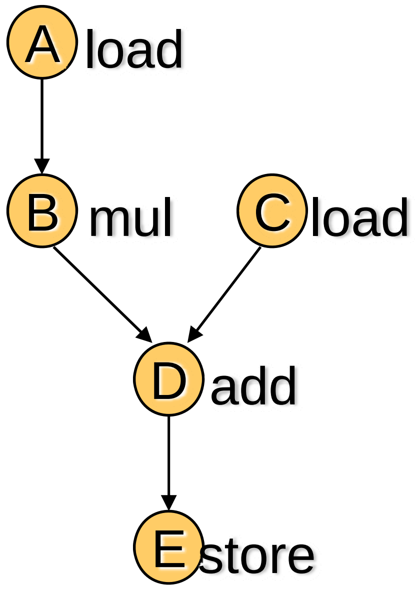


- 2 pipelined units
  - 1 MEM unit
    - load 2 cycles
    - store 1 cycle
  - 1 ALU unit
    - mul 3 cycles
    - add 3 cycles

- Computing the MII
  - Resources limit the II in this example
    - MEM = 3 uses / 1 resource
    - ALU = 2 uses / 1 resource
  - MII = 3

$$resMII = \max_i \left(\left\lceil \frac{resUssage_i}{resCount_i} \right\rceil \right) = \max \left(\left\lceil \frac{2}{1}, \frac{3}{1} \right\rceil \right) = 3$$

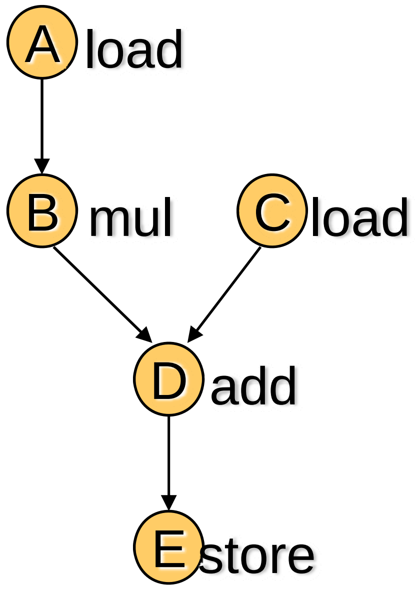
Modulo Scheduling example



Stage	Cycle	MEM	ALU
0	0	A	
	1		
	2		
1	3		
	4		
	5		
2	6		
	7		
	8		
3	9		
	10		
	11		

MRT		
Cycle	MEM	ALU
0	X	
1		
2		

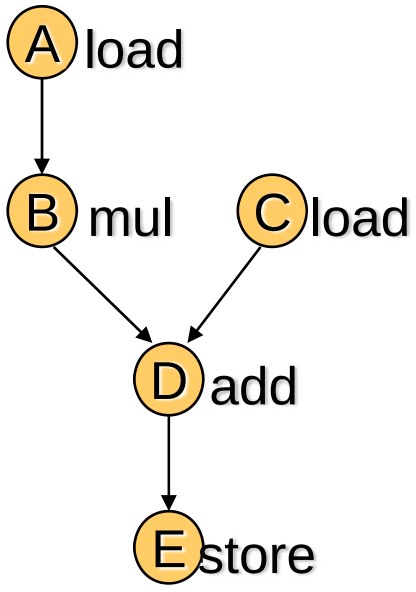
Modulo Scheduling example



Stage	Cycle	MEM	ALU
0	0	A	
	1		
	2		**B**
1	3		
	4		
	5		
2	6		
	7		
	8		
3	9		
	10		
	11		

MRT		
Cycle	MEM	ALU
0	X	
1		
2		**X**

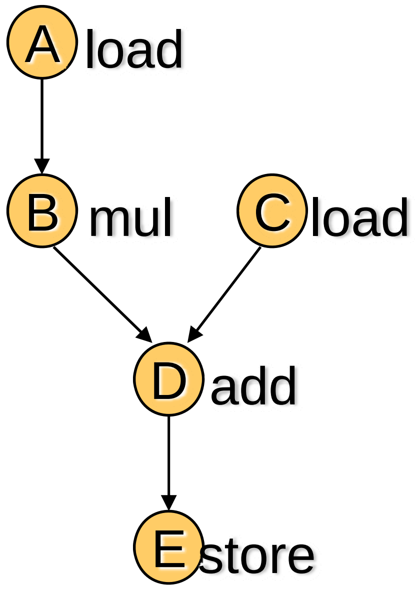
Modulo Scheduling example



Stage	Cycle	MEM	ALU
0	0	A	
	1	**C**	
	2		B
1	3		
	4		
	5		
2	6		
	7		
	8		
3	9		
	10		
	11		

MRT		
Cycle	MEM	ALU
0	X	
1	**X**	
2		X

Modulo Scheduling example

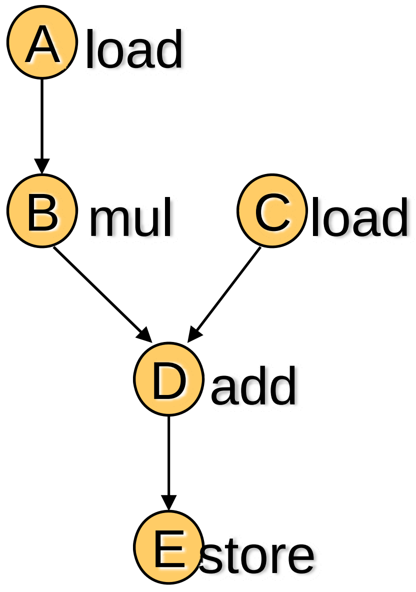


Stage	Cycle	MEM	ALU
0	0	A	
	1	C	
	2		B
1	3		
	4		
	5		D
2	6		
	7		
	8		
3	9		
	10		
	11		

MRT		
Cycle	MEM	ALU
0	X	
1	X	
2		X

Conflict!

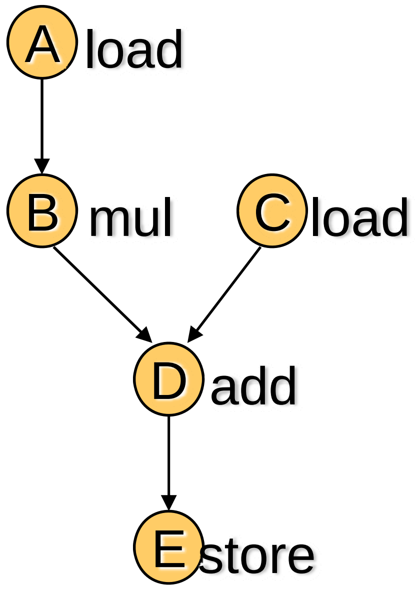
Modulo Scheduling example



Stage	Cycle	MEM	ALU
0	0	A	
	1	C	
	2		B
1	3		
	4		
	5		
2	6		**D**
	7		
	8		
3	9		
	10		
	11		

MRT		
Cycle	MEM	ALU
0	X	**X**
1	X	
2		X

Modulo Scheduling example

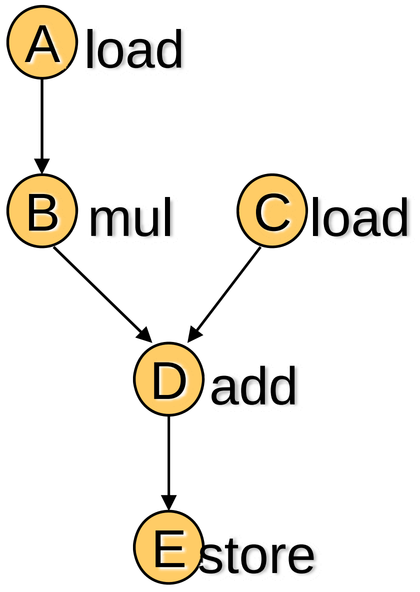


Stage	Cycle	MEM	ALU
0	0	A	
	1	C	
	2		B
1	3		
	4		
	5		
2	6		D
	7		
	8		
3	9	E	
	10		
	11		

MRT		
Cycle	MEM	ALU
0	X	X
1	X	
2		X

Conflict!!!

Modulo Scheduling example

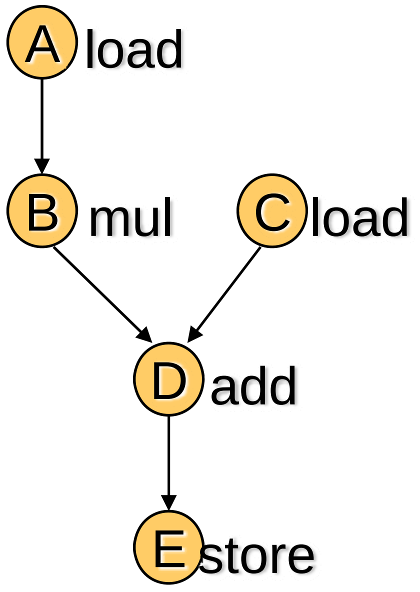


Stage	Cycle	MEM	ALU
0	0	A	
	1	C	
	2		B
1	3		
	4		
	5		
2	6		D
	7		
	8		
3	9		
	10	E	
	11		

MRT		
Cycle	MEM	ALU
0	X	X
1	X	
2		X

Conflict!!!

Modulo Scheduling example

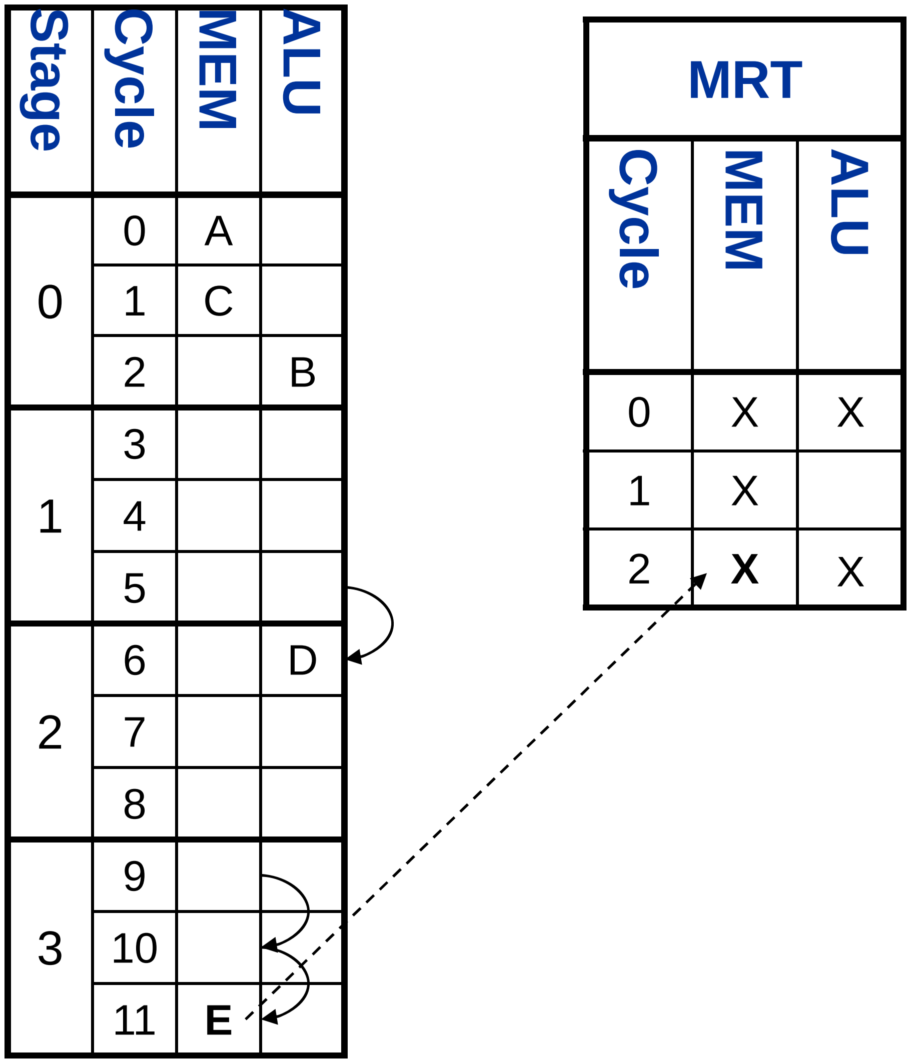


An individual iteration takes more time:

- 12 cycles
- vs 9 cycles

But N iterations take:

- 9 + 3 x N
- vs 9 x N



Stage	Cycle	MEM	ALU
0	0	A	
	1	C	
	2		B
1	3		
	4		
	5		
2	6		D
	7		
	8		
3	9		
	10		
	11	E	

MRT		
Cycle	MEM	ALU
0	X	X
1	X	
2	**X**	X

Recurrence constraints and MII

The maximum computation rate of a loop is bounded by the following ratio

$$r_{opt} = \min_C \{ Dc/Wc \}$$

where C is a dependence cycle, Dc is the total dependence distance along C and Wc is total execution time of C. i.e.

$$Dc = \sum_{e \in c} d_e$$

(d_e is the dependence distance along the edge e in C)

$$Wc = \sum_{e \in c} w_e$$

(w_e is the edge weight (latency) along the edge e in C)

Computing RecMII

$$RecMII = \max_{C}\left(\frac{\sum_{e\in C} Lat_e}{\sum_{e\in C} Dist_e}\right)$$

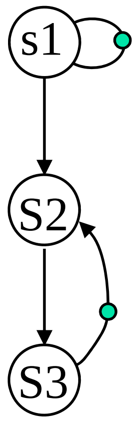
- Computing MIIrec and identifying recurrences:
  - HRMS/SMS
    - require identifying each individual recurrence circuit and sort them by criticality. An **exponential** number of recurrence circuits can appear
  - IMS & slack: MinDist Matrix

    $O(V^3) \sim O(V^4)$
  - LXMS: Marks back-edges during graph construction $O(V^2)$
    - Computes MIIrec and identifies recurrence subgraphs

      $O(V * \log V) \sim O(V^2)$

Example of recurrence constraints

```
      for i = 0 to N - 1 do
S1:           a[i] = a[i - 1] + R[i];
S2 :          b[i] = a[i] + c[i - 1];
S3 :          c[i] = b[i] + 1;
      end;
```



Note: We use a token here to represent a flow dependence of distance = 1

iterations

time	i = 0	i = 1	i = 2	i = 3
0	S1			
1	S2			...
2	S3	S1		
3		S2		
4		S3	S1	

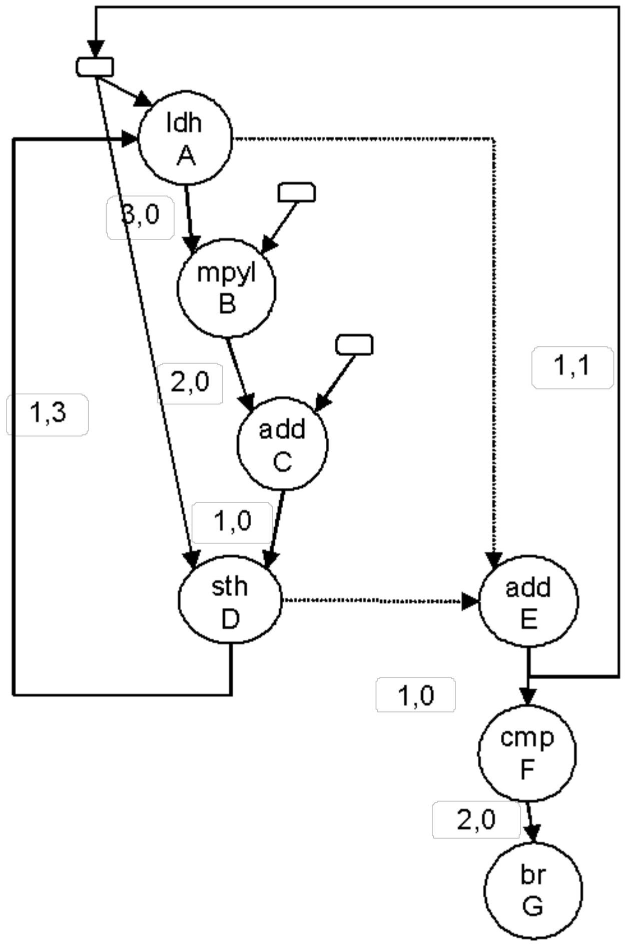
II

time

So, **RecMII** = 2

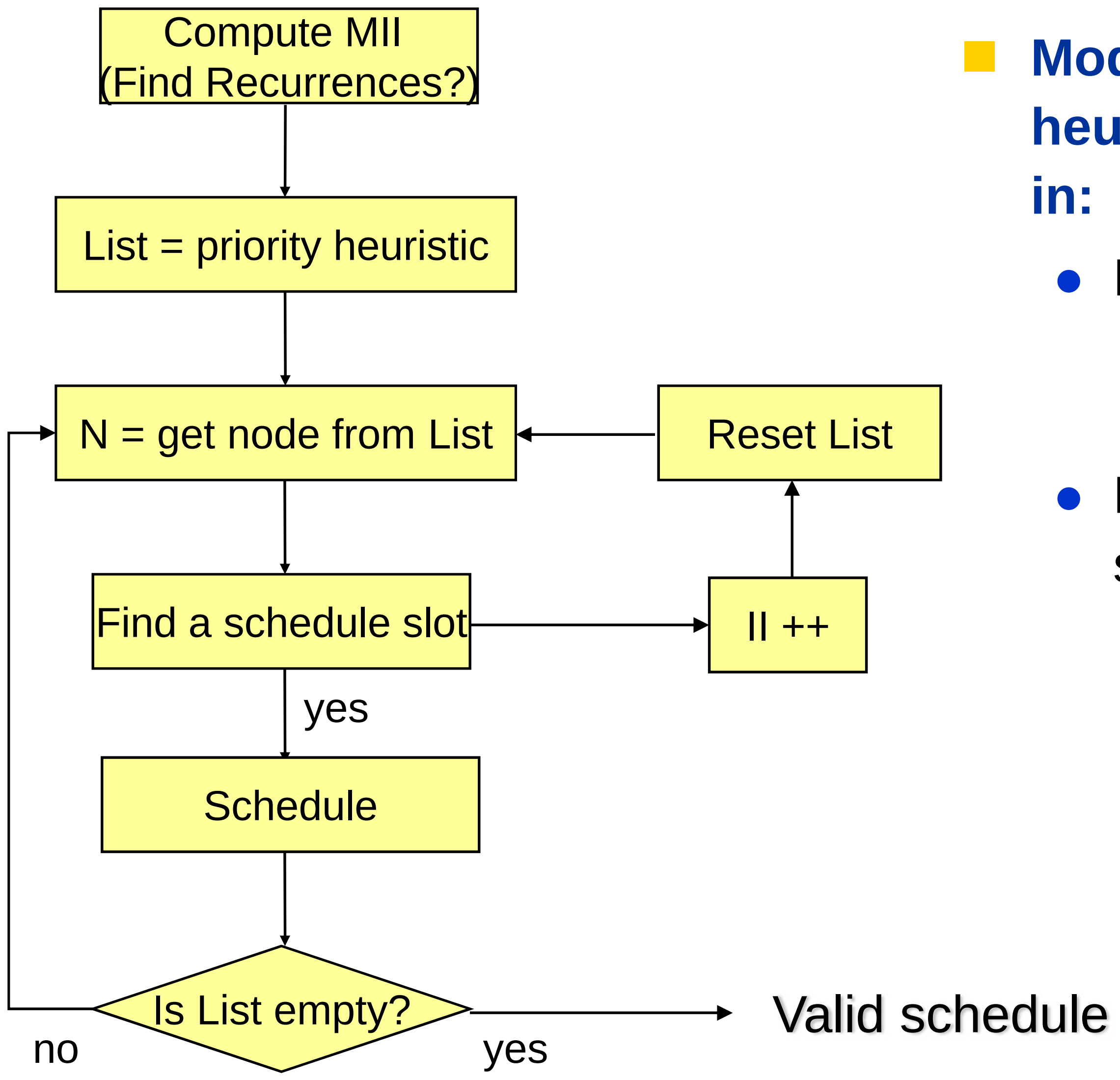
Assume each operation takes 1 cycle!

Class Problem: compute MII



- **Arcs represent**
  - latency, distance
- **Functional Units**
  - 1 MEM unit
    - ✓ ldh
    - ✓ sth
  - 4 ALU units
    - ✓ add
    - ✓ cmp
    - ✓ br
  - 2 MULT units
    - ✓ mpy

Basic modulo scheduling algorithm

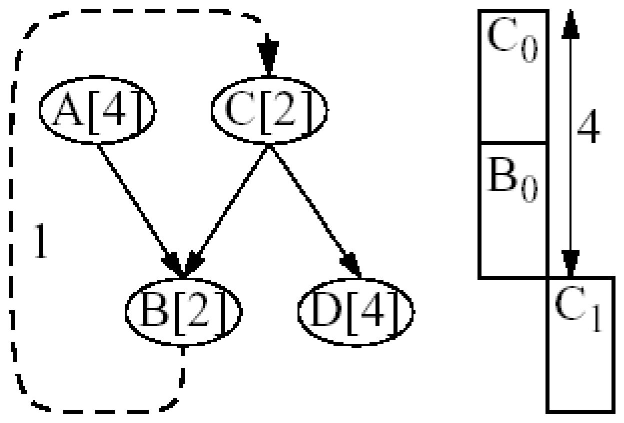


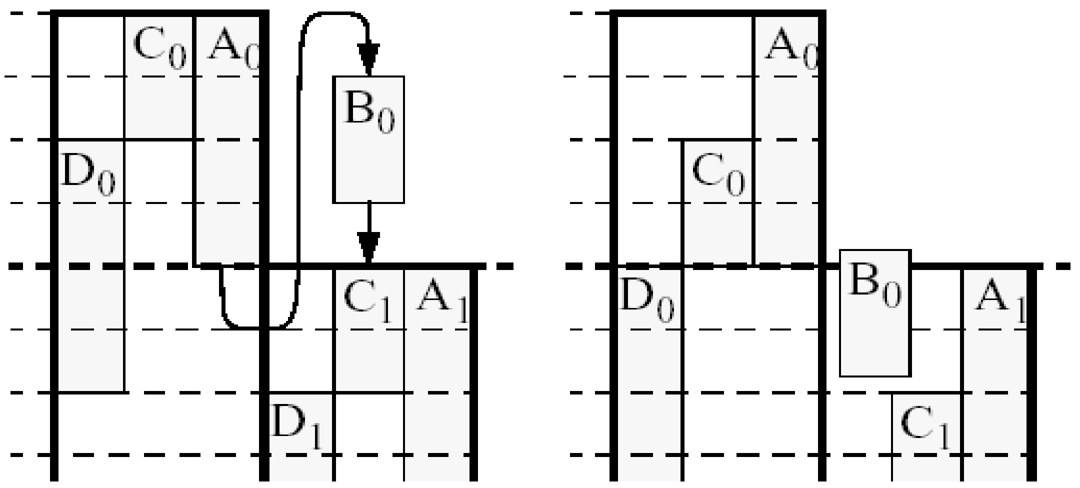
- **Modulo scheduling heuristics mostly differ in:**
  - Priority heuristic
    - ✓ E.g. adapted list scheduling
  - Finding a schedule slot

Additional difficulties for MS

- Recurrence constraints difficult the schedule
- Operations with complex reservation tables
  - Non-pipelined functional units (block reservation tables)
  - Operations that use multiple resources or units
- Those constraints combined may make it impossible to achieve the MII
  - Thus MII is only a lower bound

Recurrence constraints

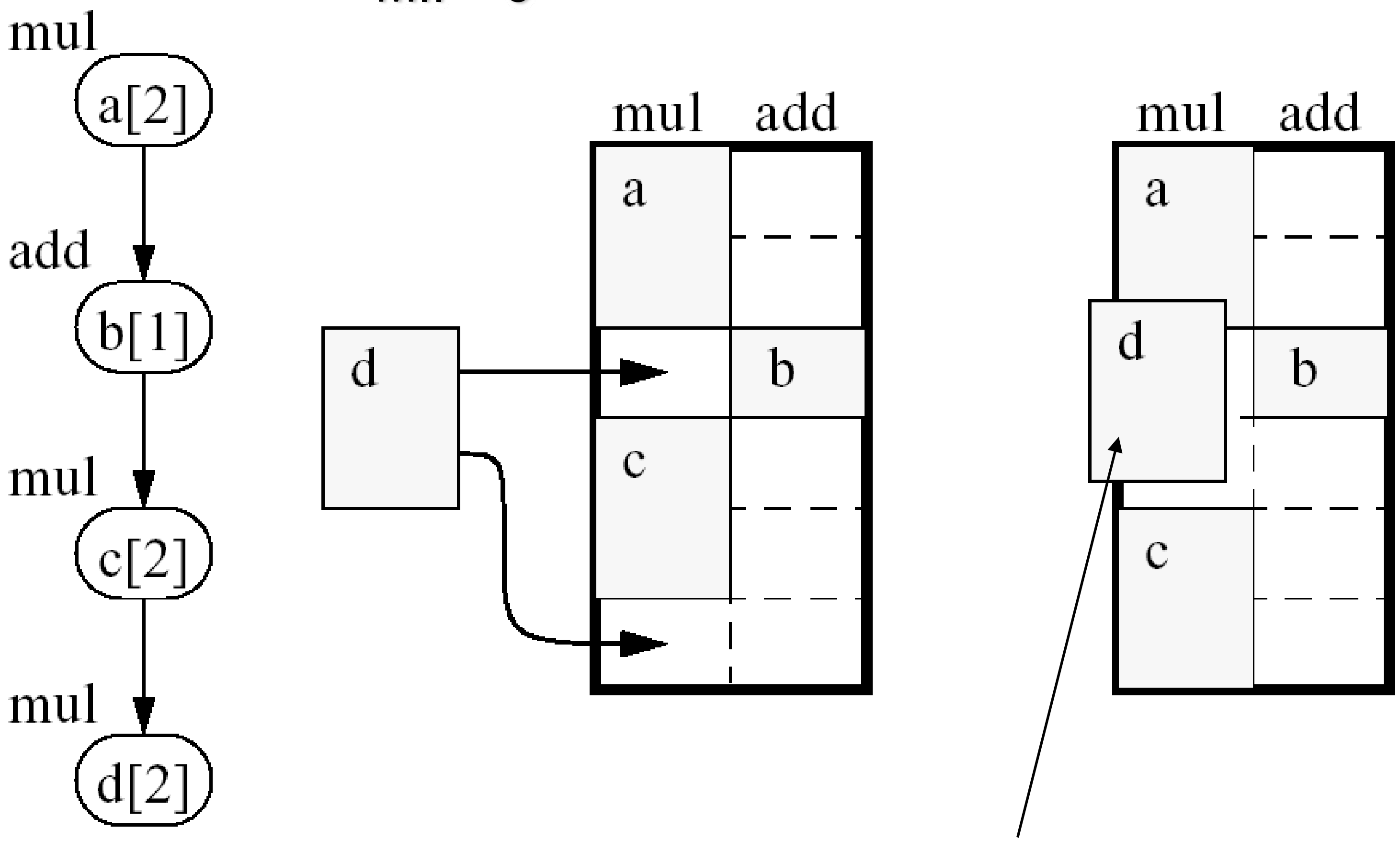




- Assume operations fully pipelined and 3 functional units
  - MII = 4 (recurrences)
- Recurrences difficult the scheduling process
  - A greedy list scheduling might not work
  - Scheduling order {A,C,D,B}
    - No space for B
  - Delaying C by 2 cycles makes space for B

Block reservation tables

MII = 6



Use 2 slots in next stage

Complex reservation tables

It is impossible to achieve II = MII =4

Optimum II = 5

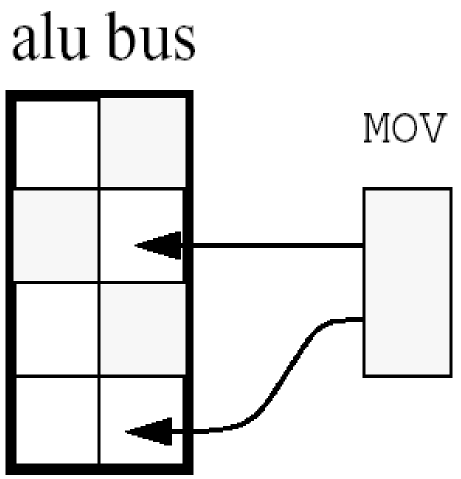
ADD

alu	bus
	X
X	
	X

MOV

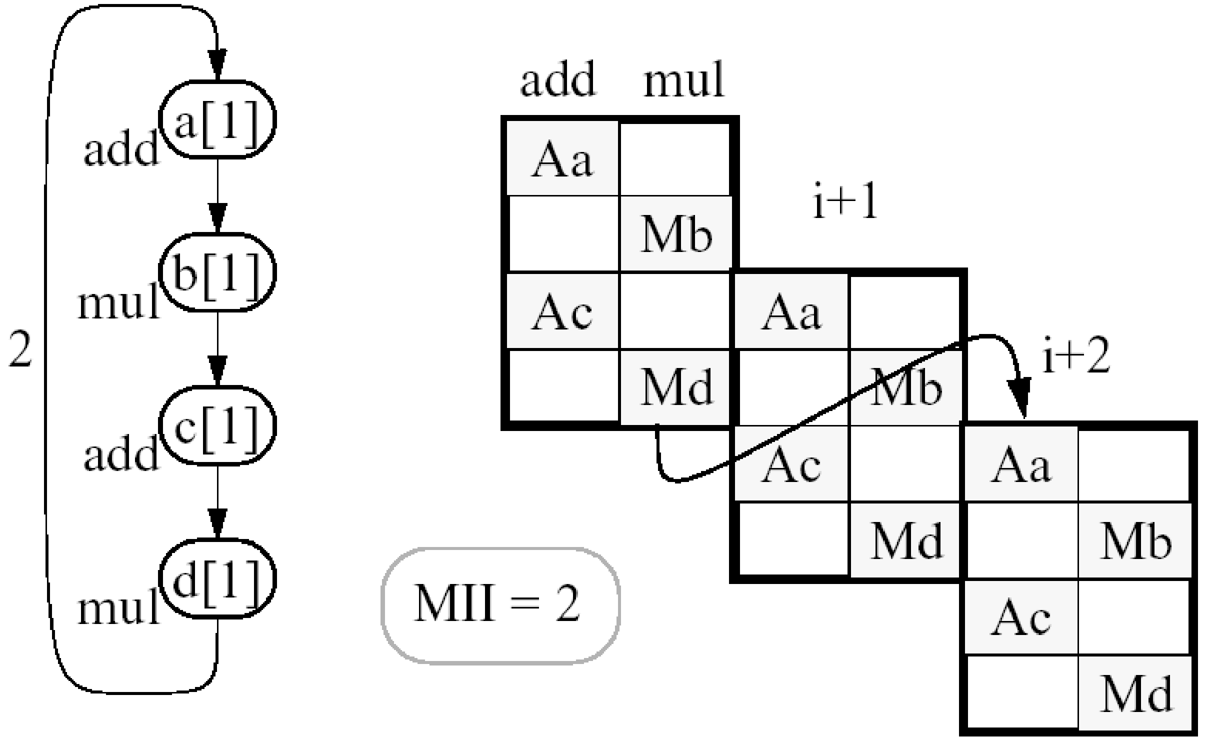
alu	bus
	X
	X

MII = 4



Resource + recurrence constraints

- Scheduling all OPs with minimum delay, resource constraints are violated
- Delaying some OP violates the recurrence



Iterative Modulo Scheduling

- Tries to solve all constraints by using a limited form of backtracking
  - If an operation does not fit in the schedule:
    - Choose a schedule slot
    - Un-schedule all conflicting operations -> go back to pending List
    - And Schedule
- A popular misconception is that IMS is slow
  - Operations are scheduled & unscheduled more times
  - However, in practice scheduling time is reduced:
    - Many loops do not require un-scheduling
    - For most loops that do, a few un-scheduled operations may save from rescheduling the whole loop with increased II
      - Faster and better performance
  - Of course some loops will consume its un-scheduling budged and end up increasing the II anyway
    - the amount of un-scheduling must be limited to avoid excessive compilation time

Priority Function

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

Acyclic:

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

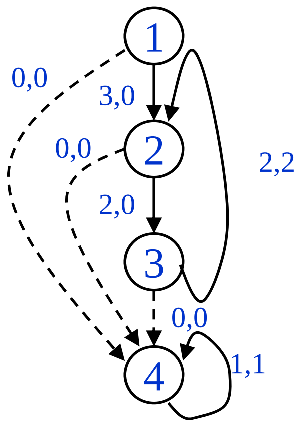
Cyclic:

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

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

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)



The Scheduling Window

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

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

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.

Latest schedule time$(Y) = L(Y) = E(Y) + II - 1$

Modulo Scheduling - Driver

```
compute MII
II = MII
budget = BUDGET_RATIO * number of ops
while (schedule is not found) do
   iterative_schedule(II, budget)
   II++
```

- Budget_ratio is a measure of the amount of backtracking that can be performed before giving up and trying a higher II
  - Empirical results show that ideal values are around 1.5-2
  - Ideal varies with architecture and benchmarks

Modulo Scheduling – Iterative Scheduler

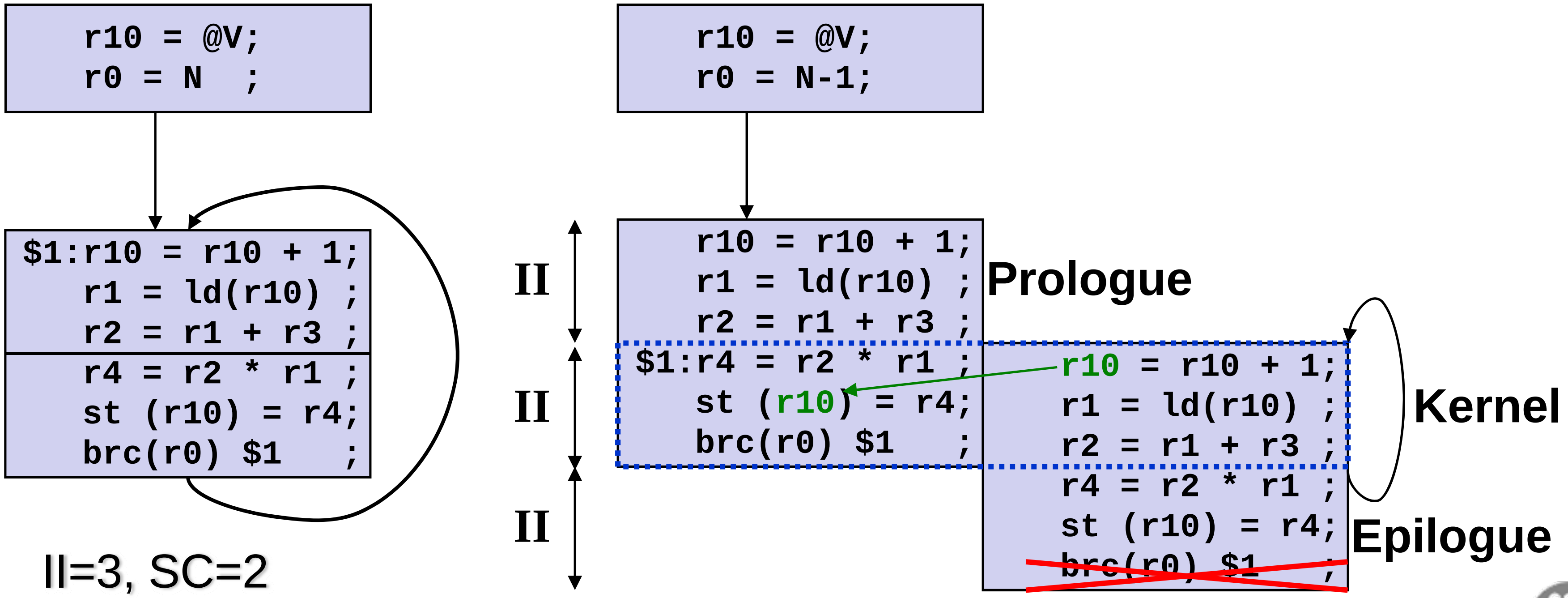
```
iterative_schedule(II, budget)
   compute op priorities
   while (there are unscheduled ops and budget > 0) do
      op = unscheduled op with the highest priority
      min = early time for op (E(Y))
      max = min + II – 1
      t = find_slot(op, min, max)
      schedule op at time t
         /* Backtracking phase – undo previous
           scheduling decisions */
         Unschedule all previously scheduled ops that
           conflict with op
      budget--
```

Modulo Scheduling – Find_slot

```
find_slot(op, min, max)
   /* Successively try each time in the range */
   for (t = min to max) do
      if (op has no resource conflicts in MRT at t)
         return t
   /* Op cannot be scheduled in its specified range */
   /* So schedule this op and displace all conflicting
     ops */
   if (op has never been scheduled or min > previous
     scheduled time   of op)
      return min
   else
      return MIN(1 + prev scheduled time of op, max)
```

Code generation Schemas

- Once a vaild schedule is found code must be generated
  - Prologue code
  - Kernel code
  - Epilogue code
- Intuitively easy to generate
  - Just overlap the schedule for SC (Stage Count) iterations
- Example is incorrect as is
  - R10 is rewritten by iteration (i+1) before using value generated in iteration (i)
  - Renaming does not work since the kernel is repeatedly executed

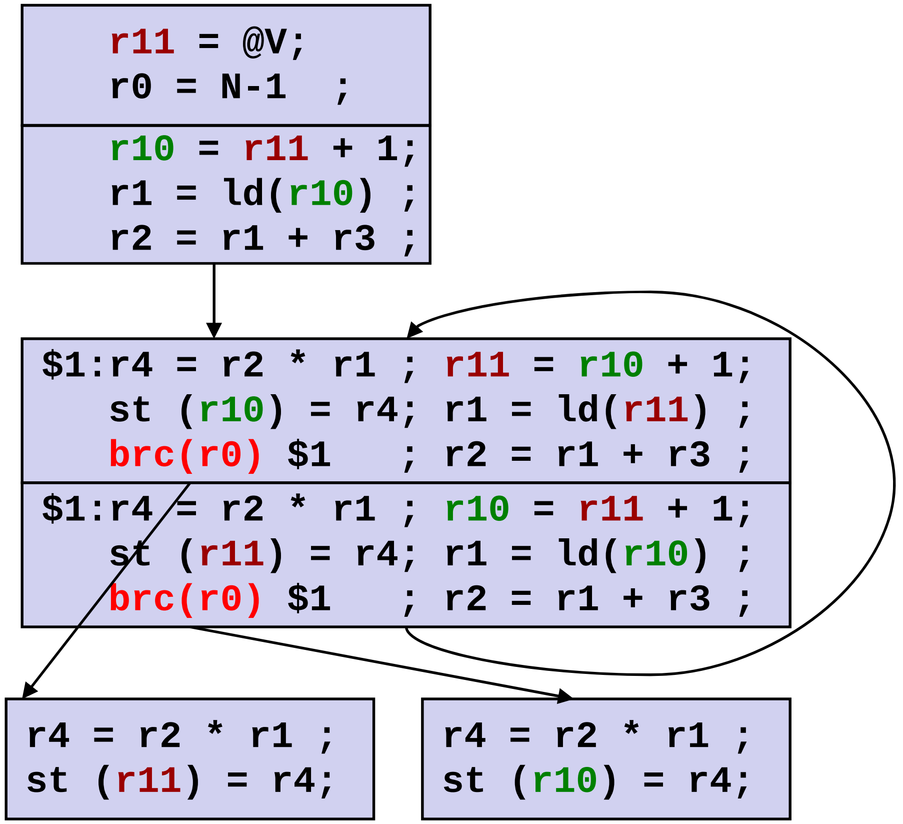


Modulo Variable Expansion

- Unroll the kernel Kmin times

$$Kmin = \max_i \left(\left\lceil \frac{Lifetime_i}{II} \right\rceil \right)$$

- Rename the overlapping registers
  - So that Def-Use chains correspond to the original iterations of the code
- Multiple versions of the epilogue are required
  - Each one uses different registers
- Noticeable code expansion
  - SC = Stage Count
  - Kmin = minimum MVE unroll
  - EBS= Exit Branch Stage



```
   r11 = @V;
   r0 = N-1   ;
```

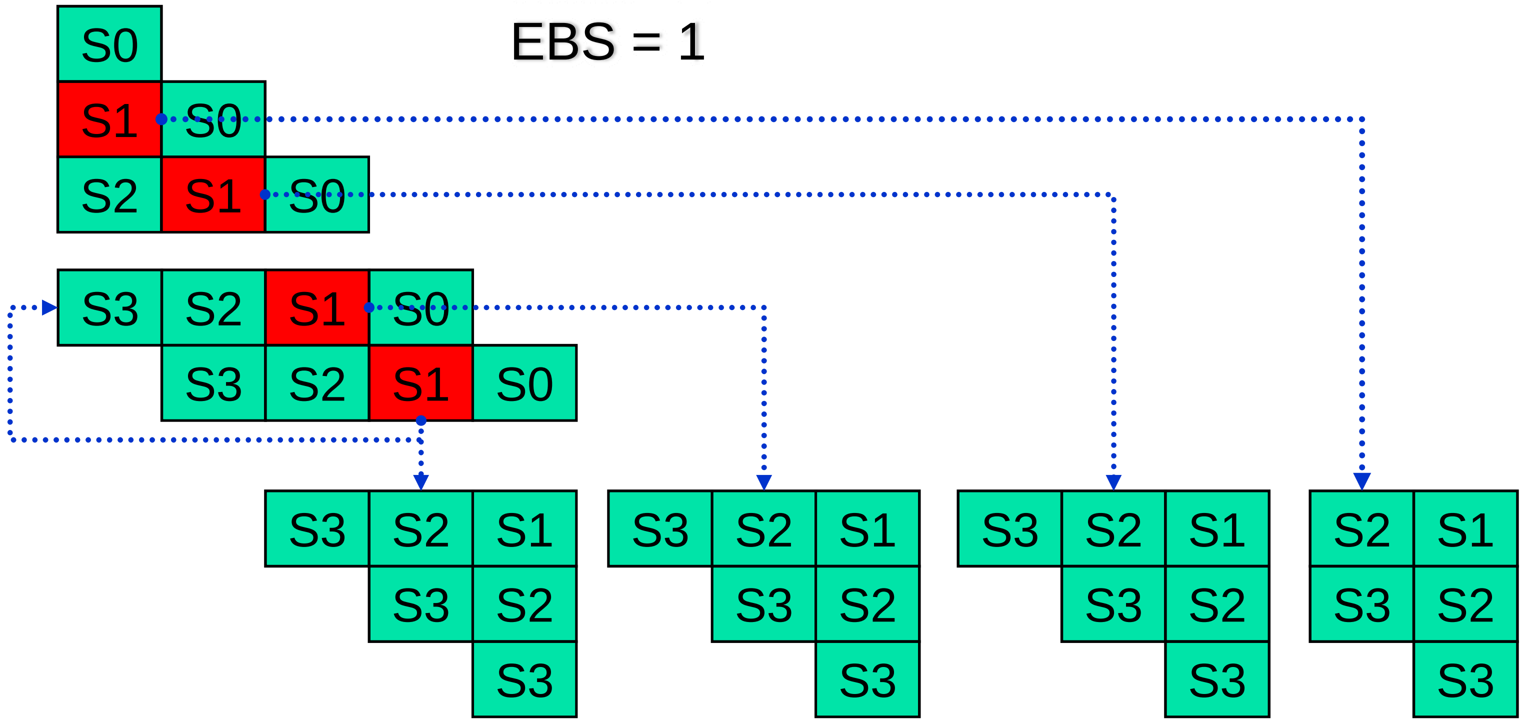
```
   r10 = r11 + 1;
   r1 = ld(r10) ;
   r2 = r1 + r3 ;
```

```
$1:r4 = r2 * r1 ; r11 = r10 + 1;
   st (r10) = r4; r1 = ld(r11) ;
   brc(r0) $1   ; r2 = r1 + r3 ;
$1:r4 = r2 * r1 ; r10 = r11 + 1;
   st (r11) = r4; r1 = ld(r10) ;
   brc(r0) $1   ; r2 = r1 + r3 ;
```

```
r4 = r2 * r1 ;
st (r11) = r4;
```

```
r4 = r2 * r1 ;
st (r10) = r4;
```

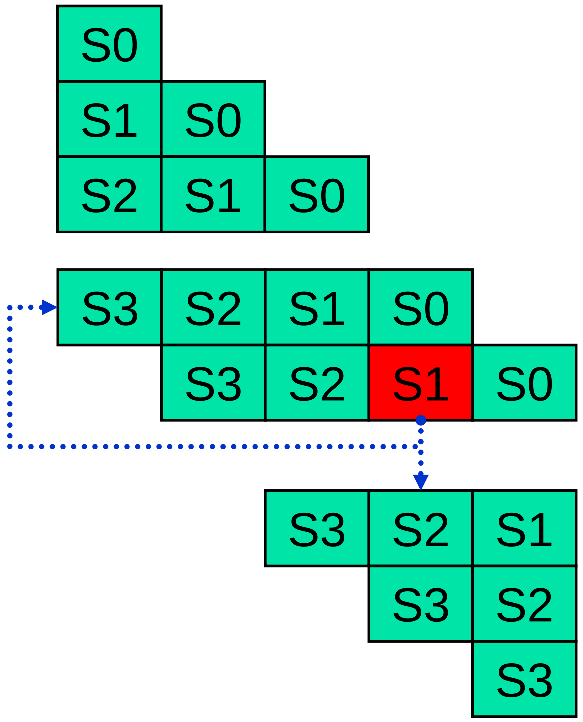
Basic code generation schema



Non Speculative code schema only for counted loops

Exit branch not necessarily scheduled in last stage

Preconditioned code schema



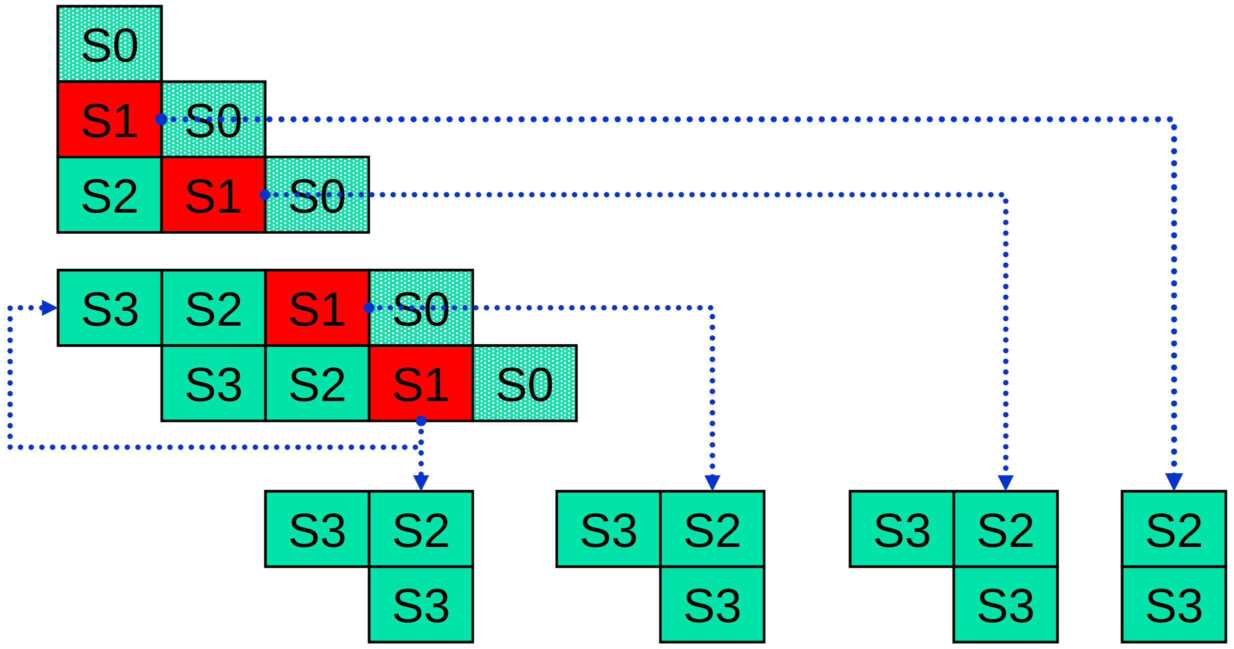
- Requires a non-pipelined preconditioning loop:
  - Execute short trip loops
  - Execute remainder iterations
- Iteration count
  - L = iterations of original loop
  - N = iterations executed by the preconditioning loop
  - M = Iterations executed by modulo scheduled loop

$$N = \begin{cases} L & \text{if } L < SC - 1 \\ \lceil L - (SC-1) \rceil \bmod Kmin & \text{otherwise} \end{cases}$$

$$M = L - N$$

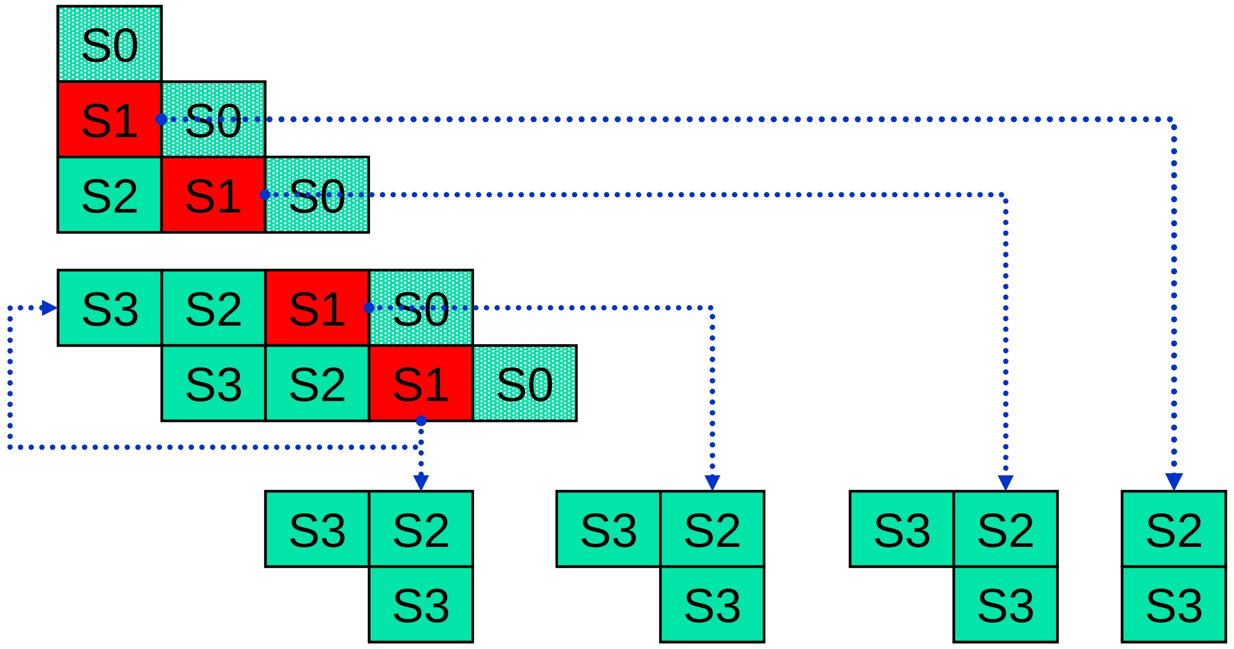
- Disadvantages
  - Adds a whole loop body to the code size.
  - Only for counted loops

Speculative code schema



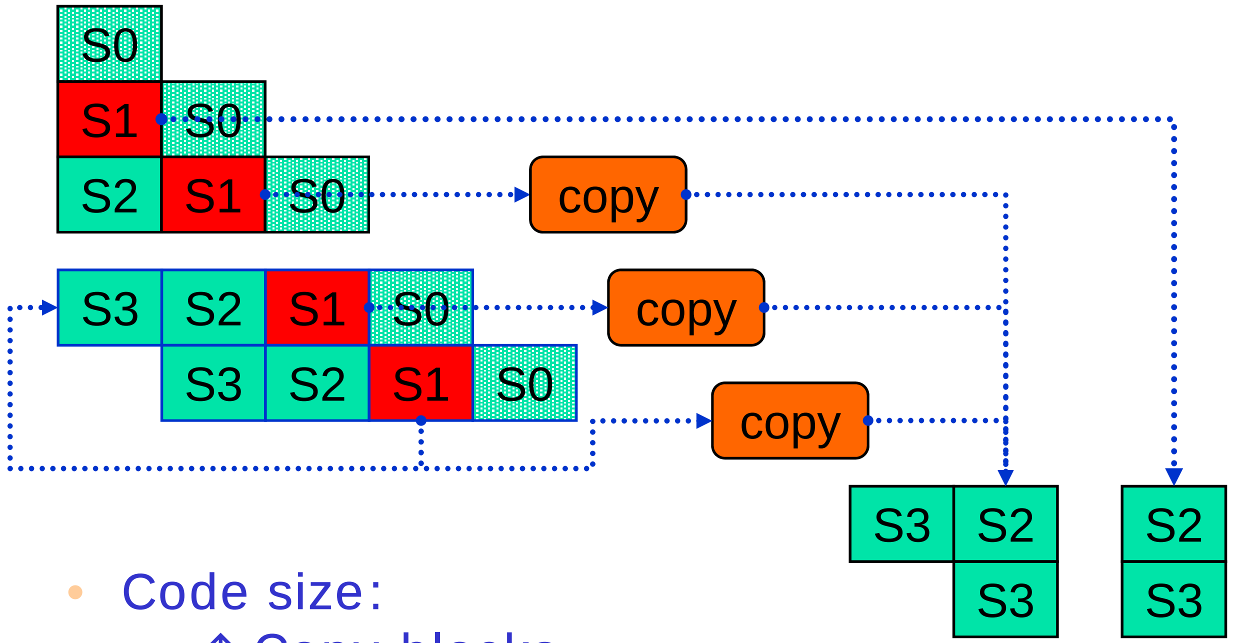
- Counted loops and loops with conditional exit.
  - [Rau et al., Micro25] [Lavery & Hwo, Micro 29]

Collapse kernel epilogs



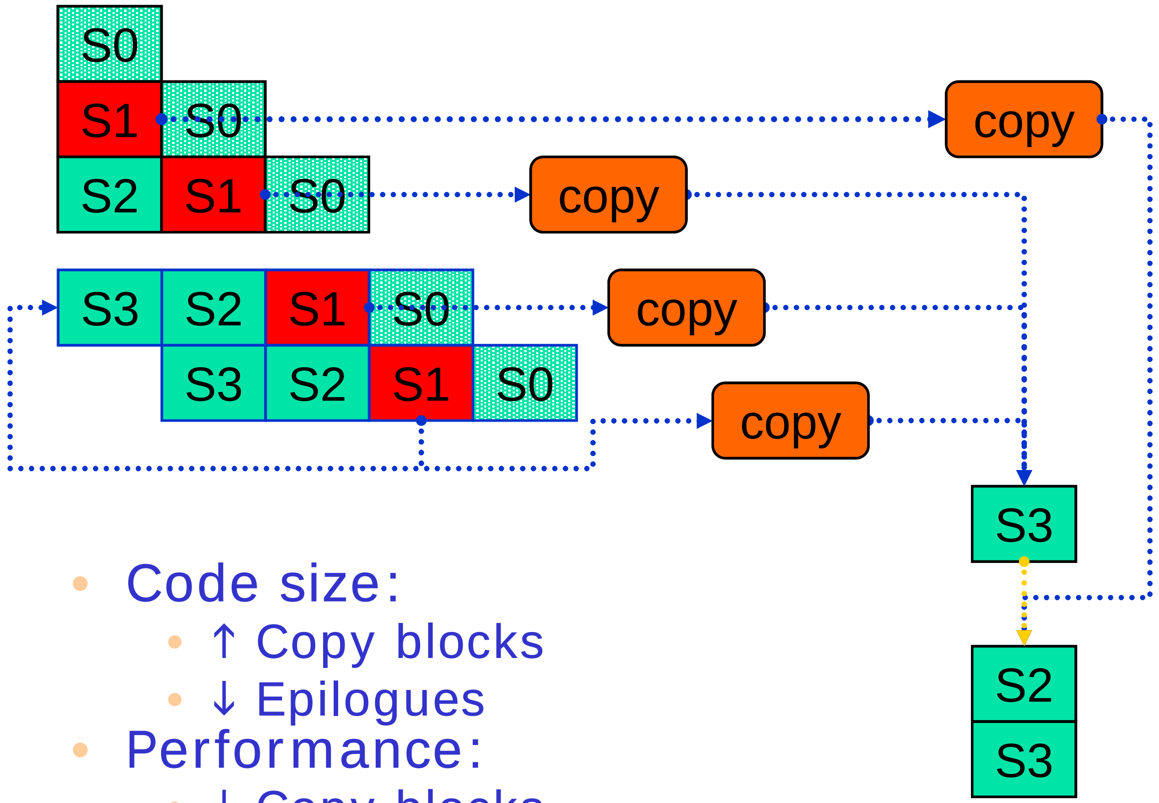
- The three kernel epilogs are identical except the registers they use
- The latter the exit branch -> more speculation -> smaller epilogues
  - Exit branch in last stage -> epilogues are empty

Collapse kernel epilogues



- Code size:
  - ↑ Copy blocks
  - ↓ Epilogues
- Performance:
  - ↓ Copy blocks
  - ↑ Epilog overlaps with next BB

Collapse prologue epilogues



- Code size:
  - ↑ Copy blocks
  - ↓ Epilogues
- Performance:
  - ↓ Copy blocks
  - ↓ Epilog is serialized
  - ↑ Last epilog BB overlaps with next BB

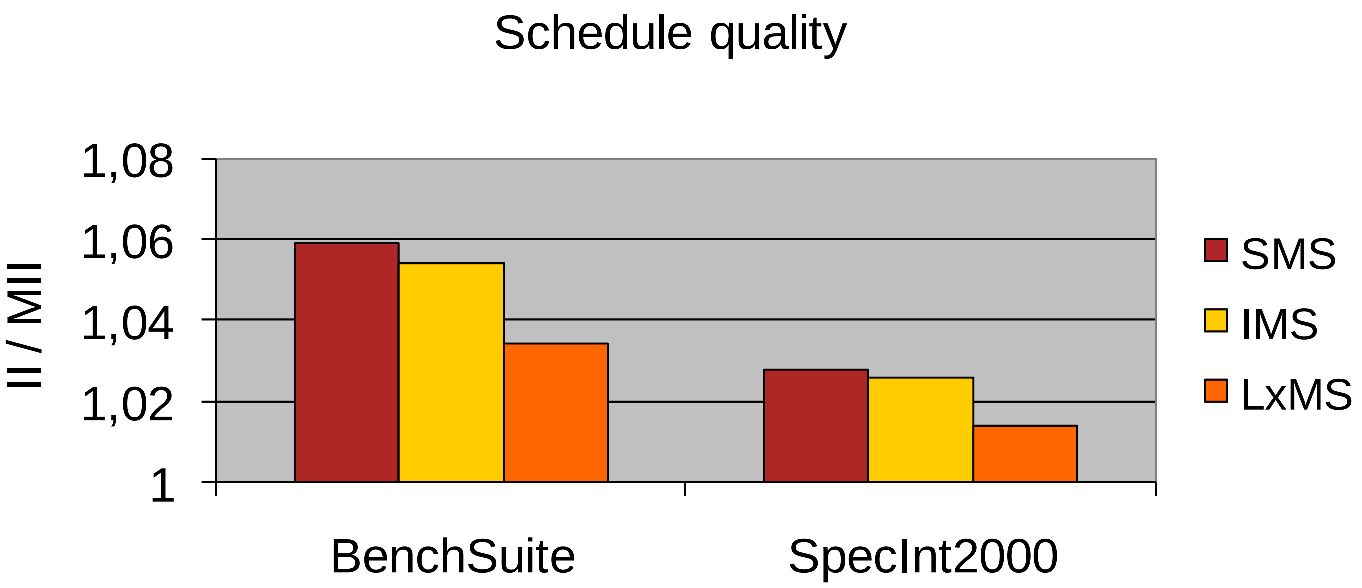
Reduce MVE by copy insertion

- Insert copies in the schedule to split lifetimes. This step is performed after scheduling using the unused slots:
  - Until MVE is reduced do:
    - Select the longest lifetime that can be split by a copy?
    - Insert copy in DDG between the producer and the last user
    - Schedule the copy starting at a midpoint cycle:

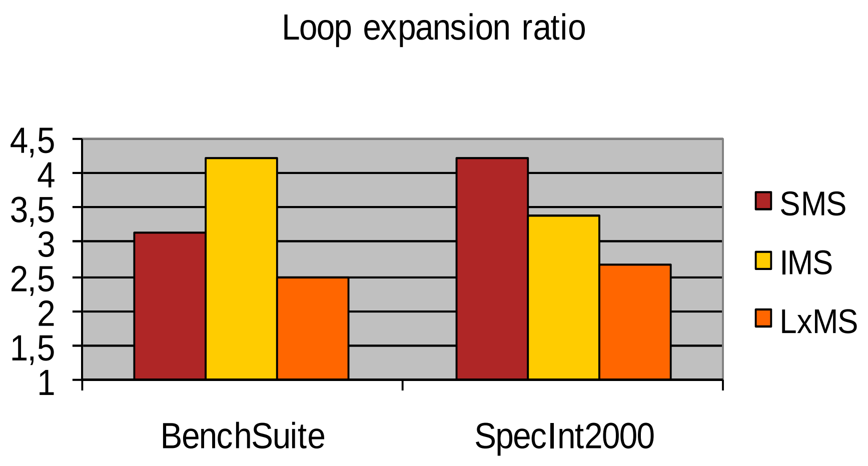
      m, m+1, m-1, m+2, m-2, ….
    - Connect all consumers that can read from the copy
  - If MVE is reduced update IR and try to further reduce MVE
  - If any step fails, remove non-necessary copies

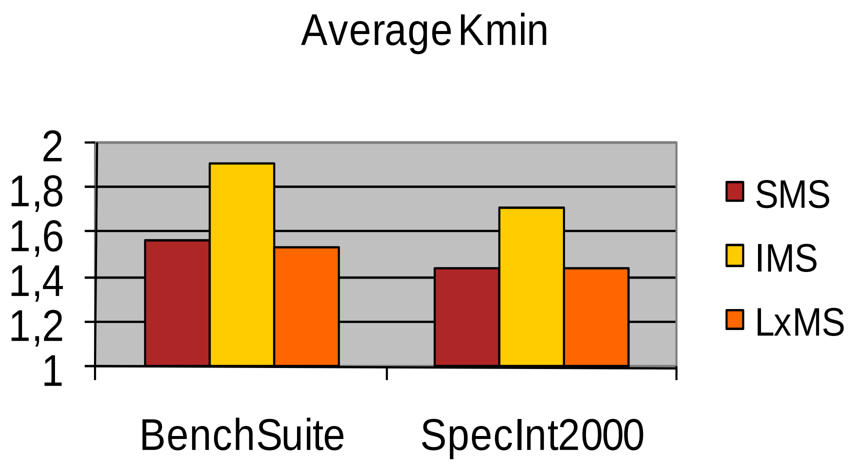
Impact of modulo scheduling heuristics

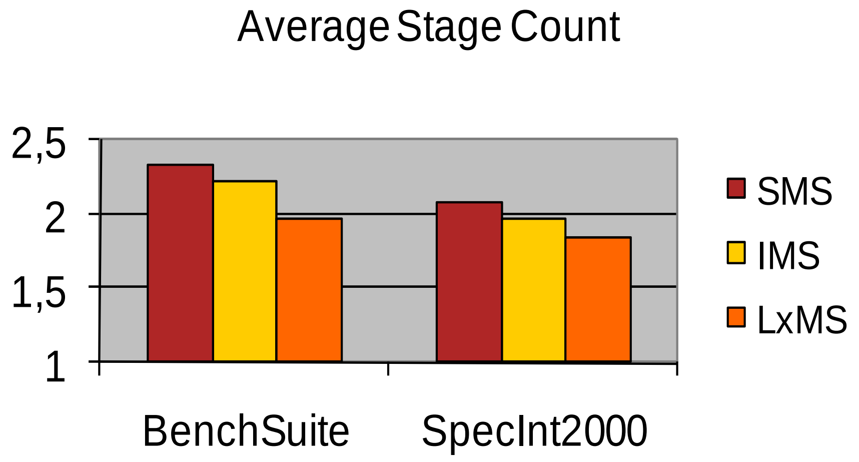
- SMS – Swing Modulo Scheduling (Llosa et al. PACT96)
  - Used only to show the influence of the modulo scheduler
- IMS – Iterative Modulo Scheduling (Rau Micro27)
  - Used as baseline
- LxMS – LxBE's modulo scheduler (Llosa, Freudenberger & Zalamea) HPLC

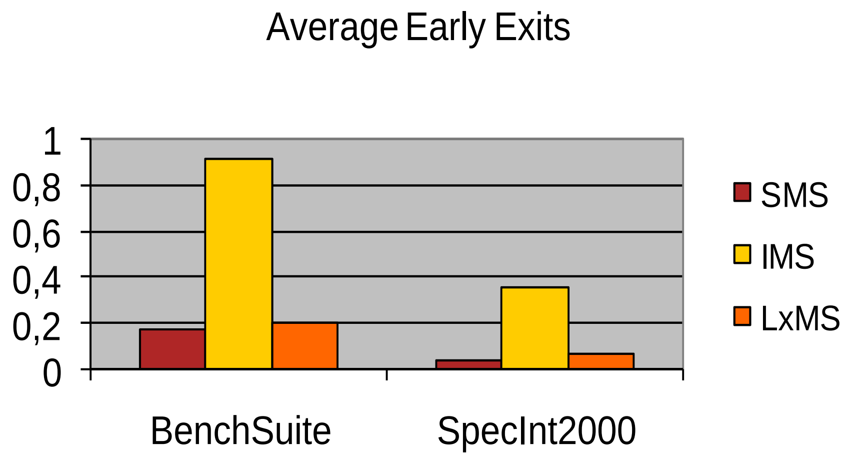


Impact of modulo scheduling heuristics

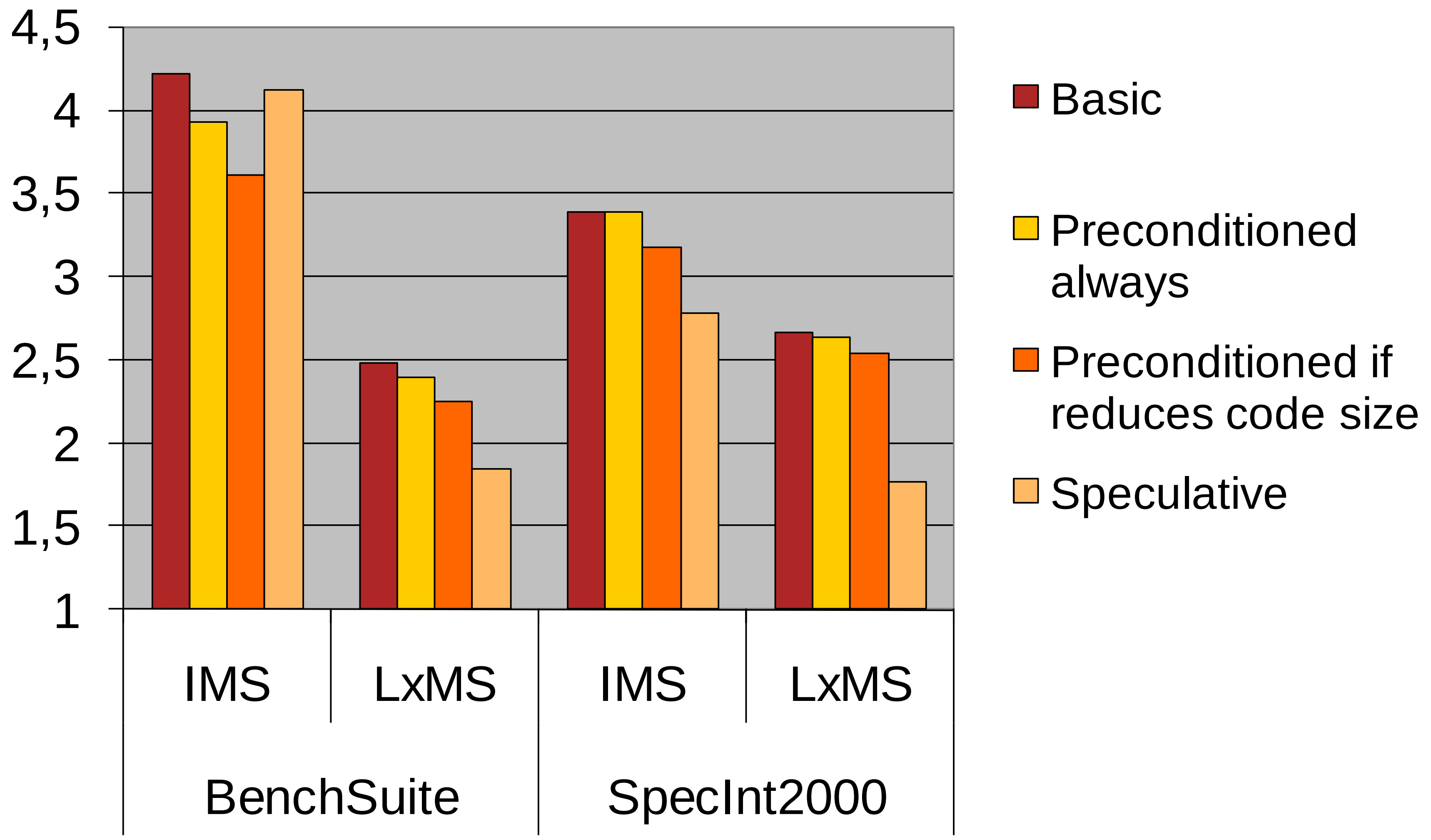




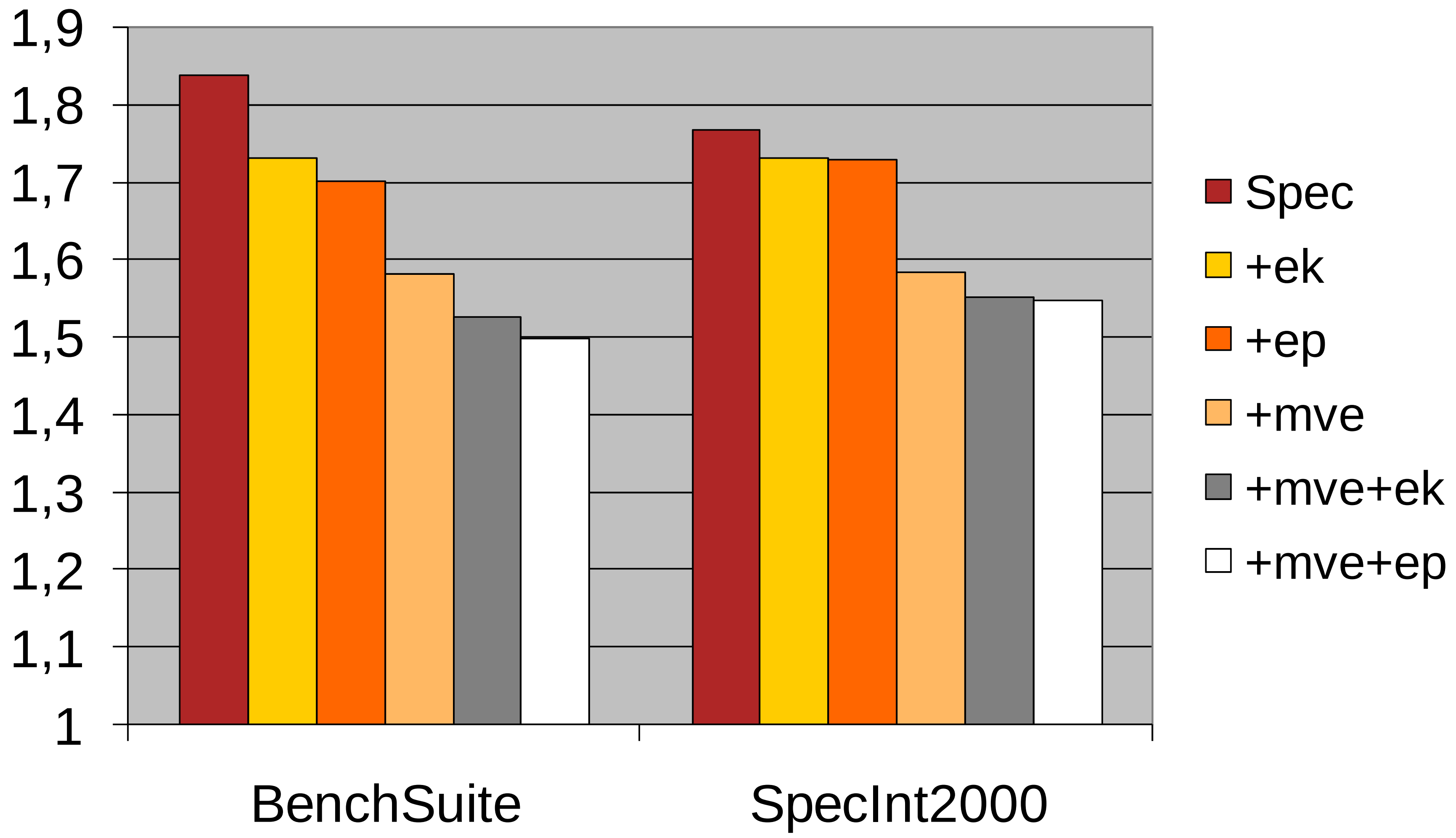




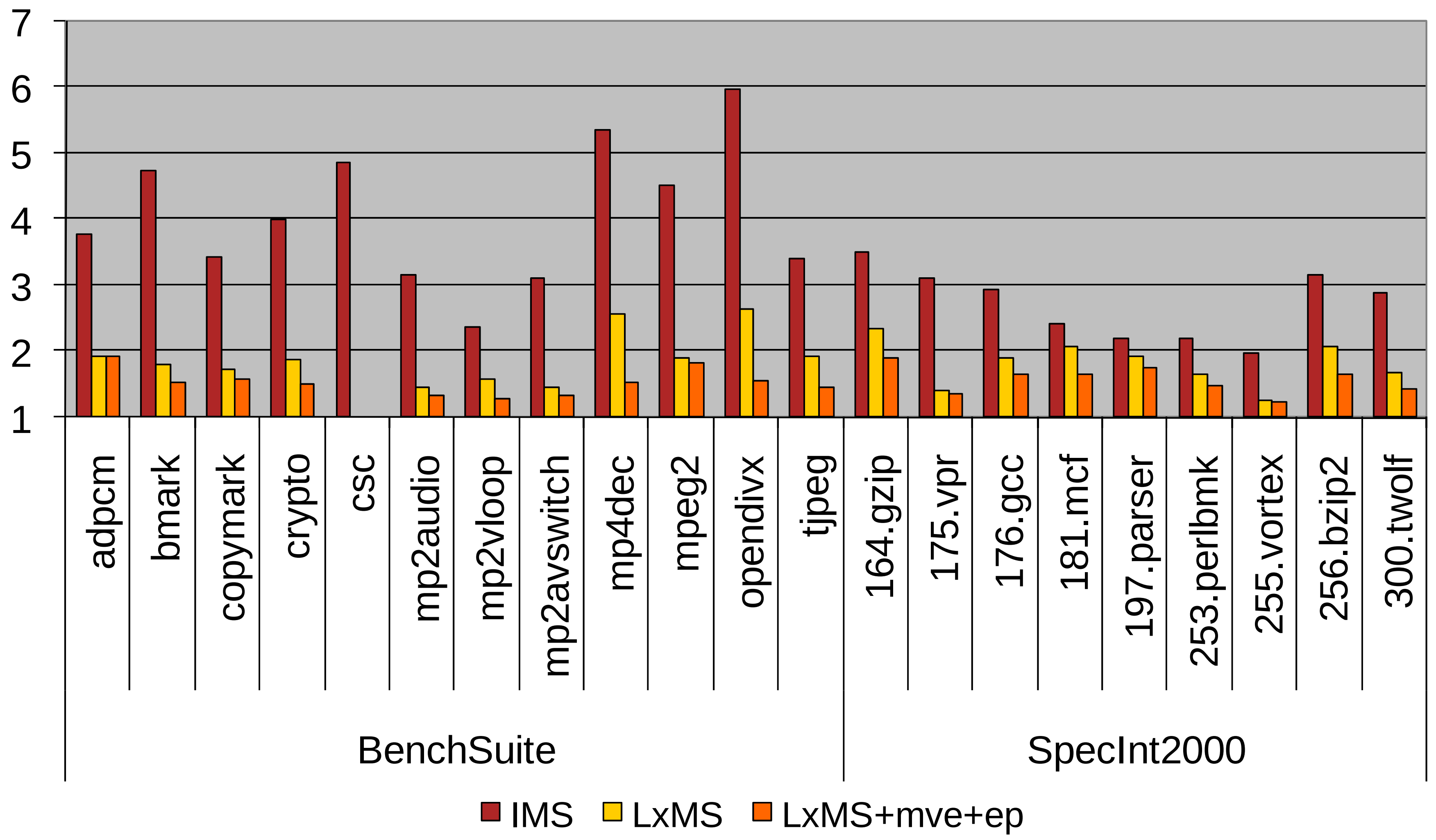
Code size expansion ratio



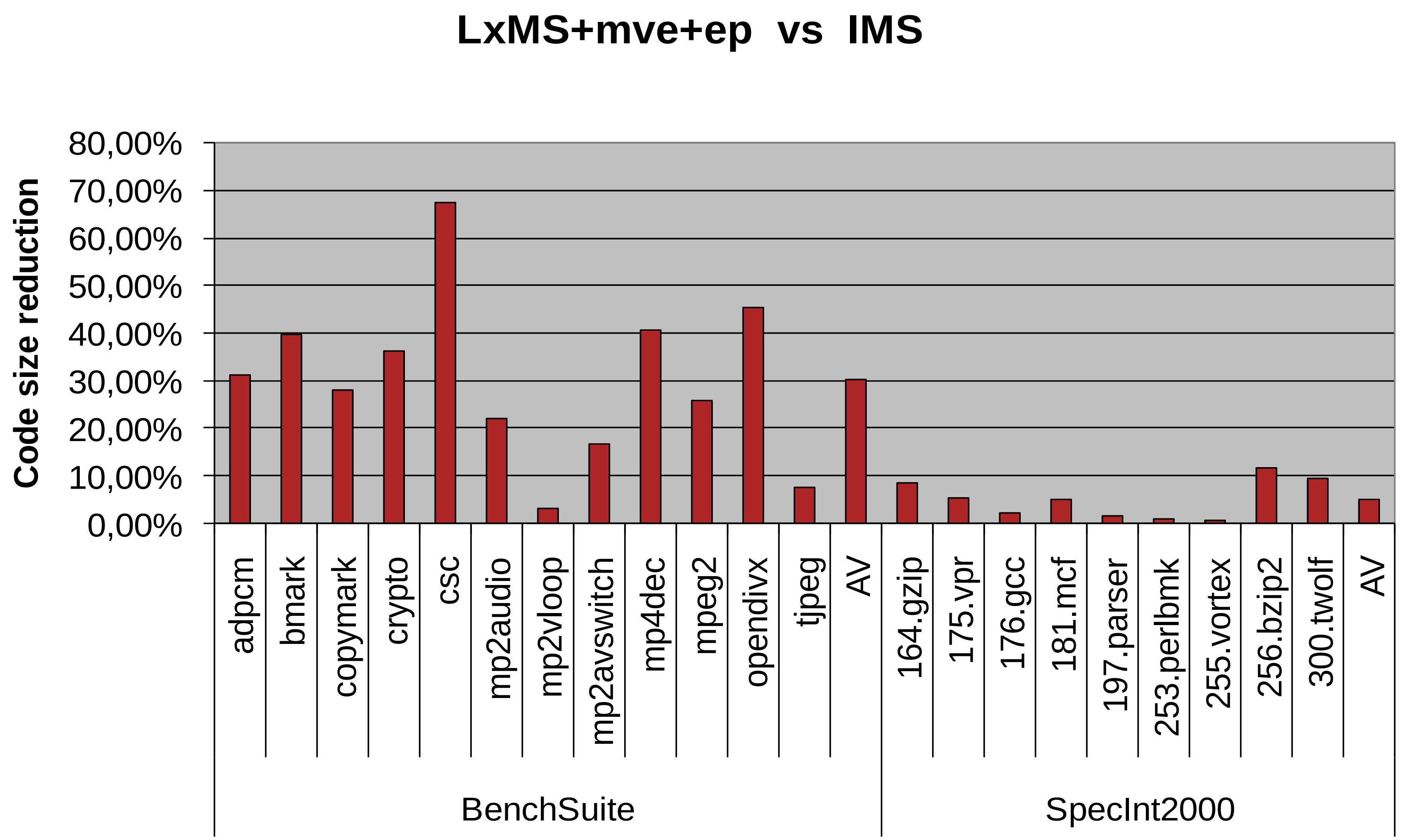
Code size reduction schemas



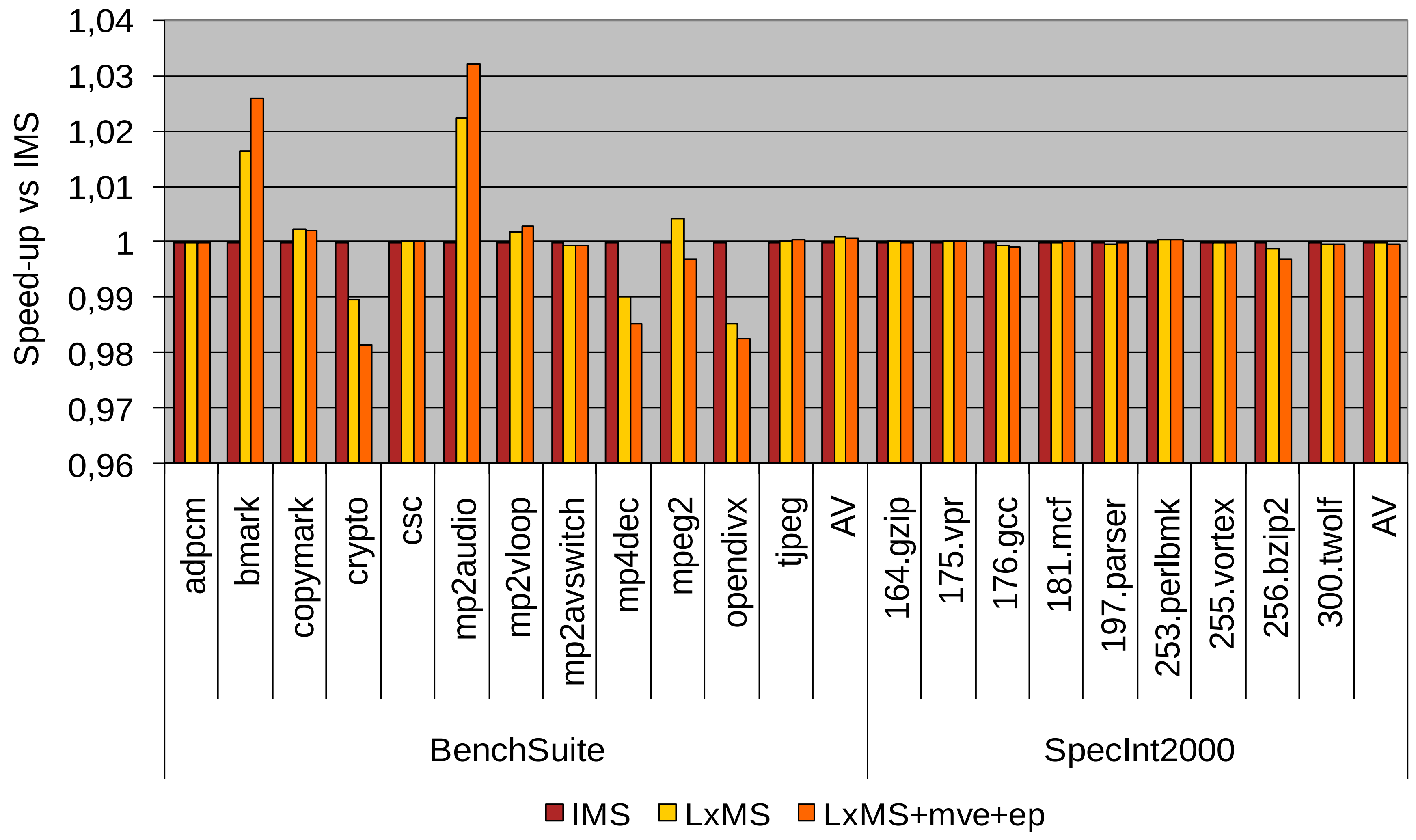
Code expansion ratio



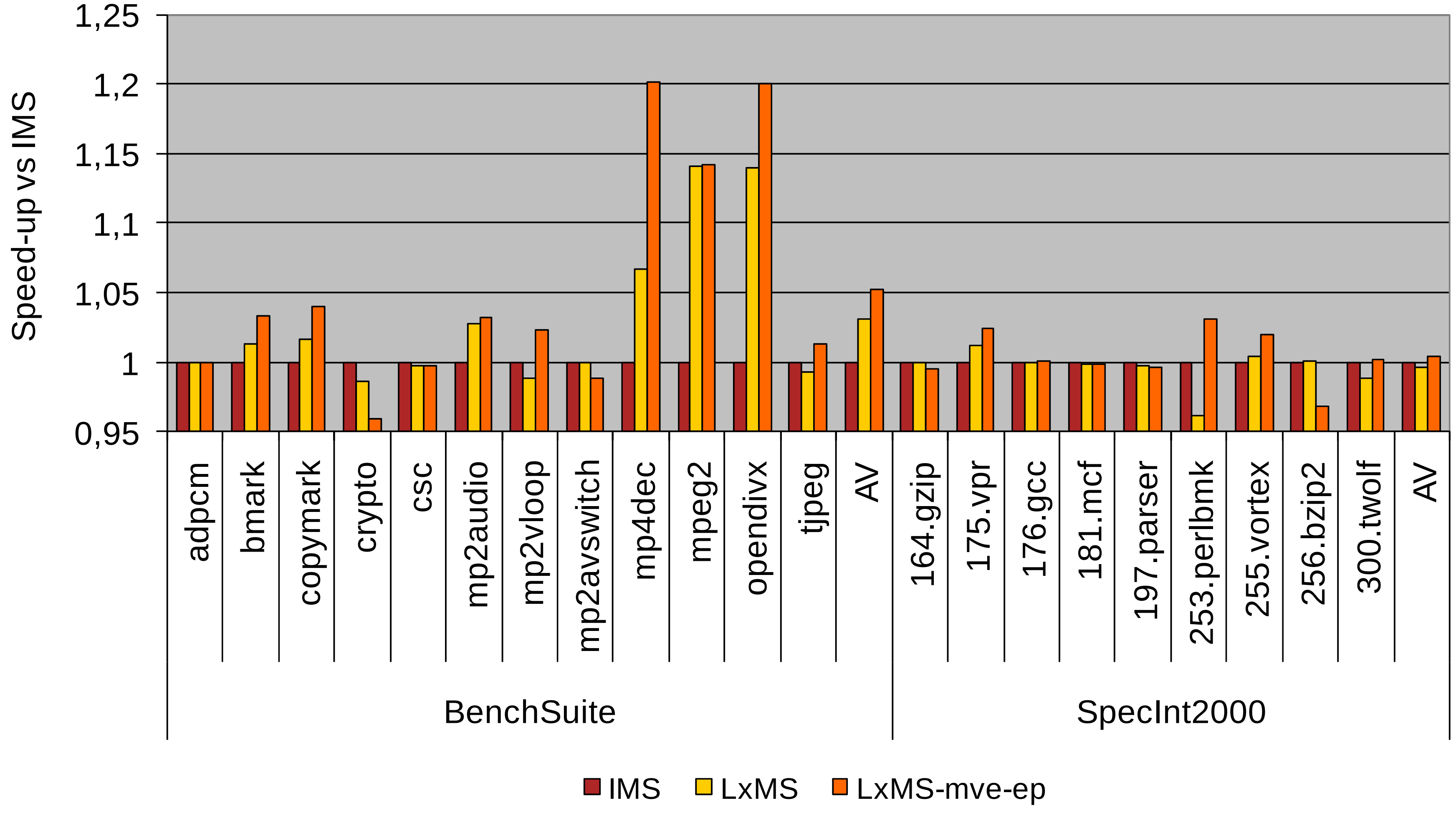
Overall code size reduction



Impact on performance w/o memory stalls



Impact on performance w. memory stalls

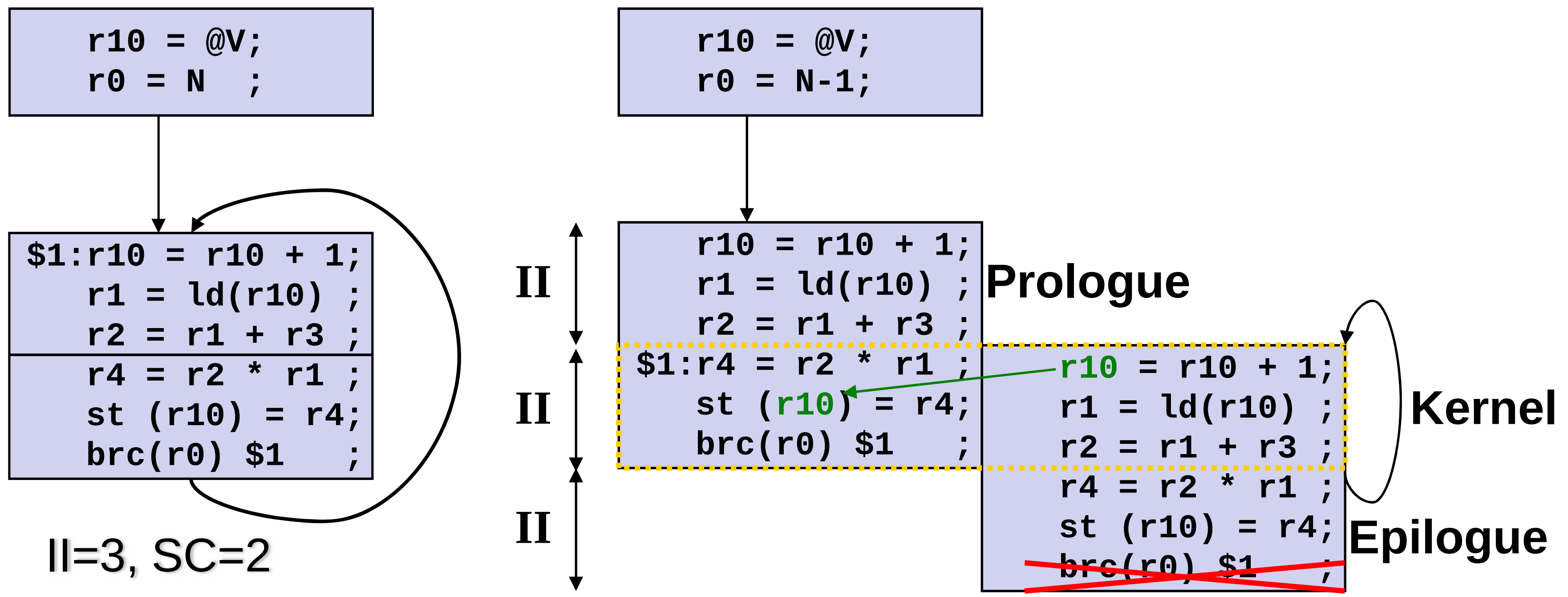


Hardware Support for Modulo Scheduling

Compilers for High Performance Architectures

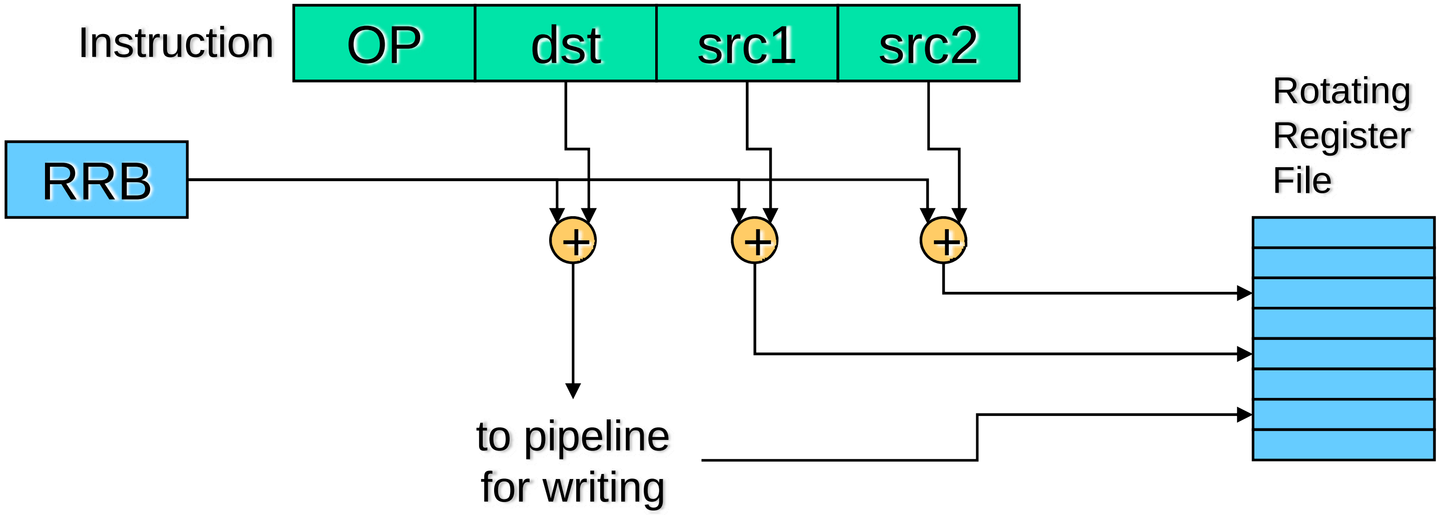
Code generation reminder

- Example is incorrect as is
  - R10 is rewritten by iteration (i+1) before using value generated in iteration (i)

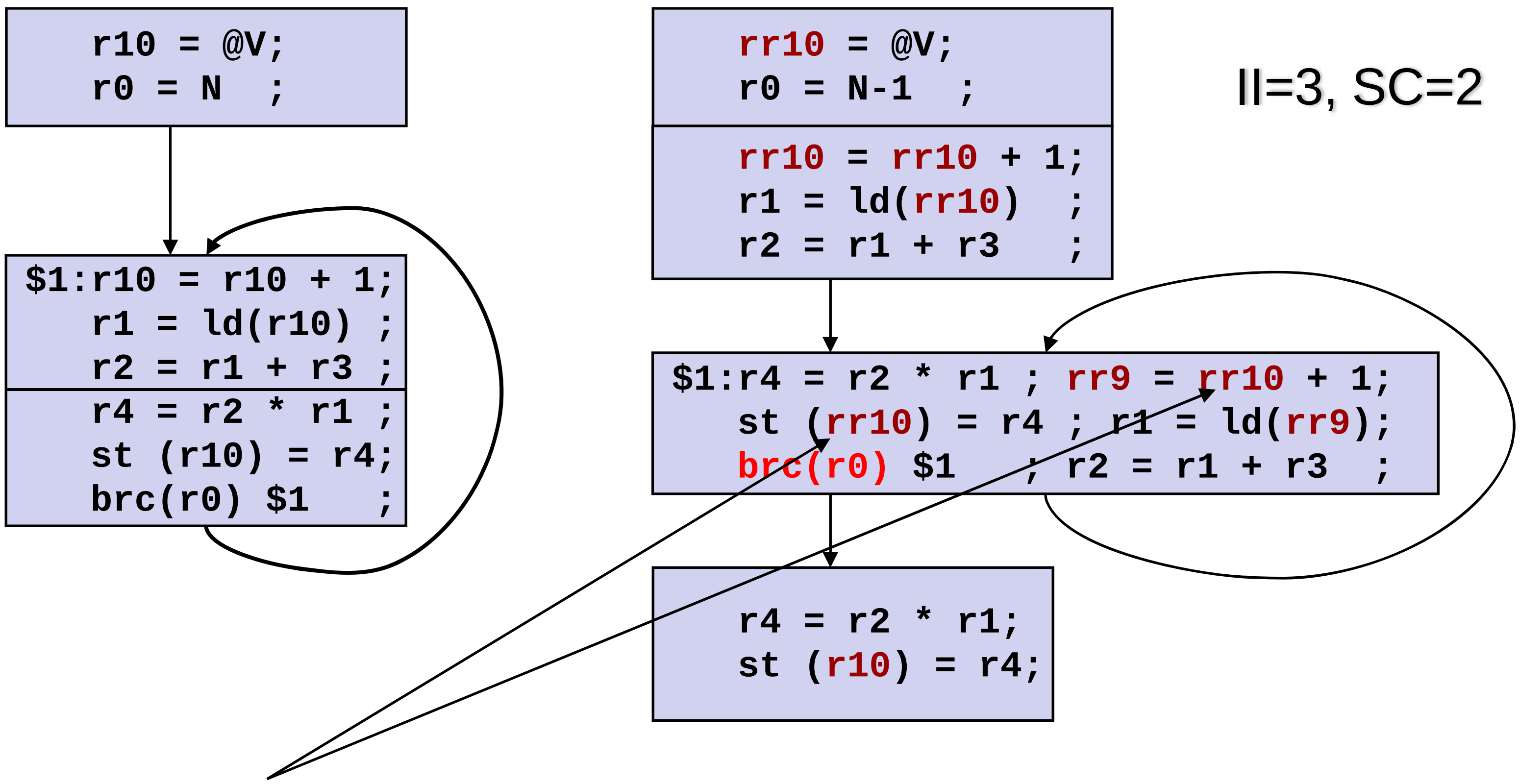


Rotating register files

- Differentiate between Logical & Physical registers
  - Contents of logical register Ri in iteration i
  - Are accessed through logical register Ri+1 in iteration i+1
- Simple renaming using a rotating registers base (RRB)
  - RP = (RRB+RL) mod NR (n-bit natural addition without carry)
    - RP = Physical register
    - RL = Logical Register
    - NR = Number of registers
  - RRB is decremented mod NR each kernel iteration



Rotating register files



```
    r10 = @V;
    r0 = N   ;
```

```
$1:r10 = r10 + 1;
   r1 = ld(r10) ;
   r2 = r1 + r3 ;
   r4 = r2 * r1 ;
   st (r10) = r4;
   brc(r0) $1   ;
```

```
    rr10 = @V;
    r0 = N-1   ;
    rr10 = rr10 + 1;
    r1 = ld(rr10)  ;
    r2 = r1 + r3   ;
```

II=3, SC=2

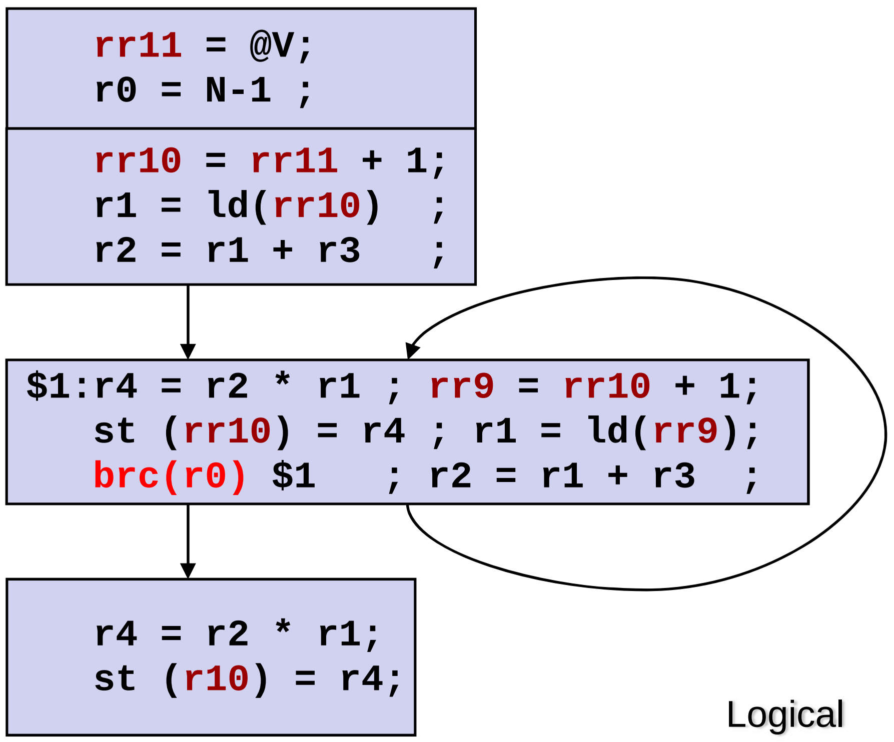
```
$1:r4 = r2 * r1 ; rr9 = rr10 + 1;
   st (rr10) = r4 ; r1 = ld(rr9);
   brc(r0) $1     ; r2 = r1 + r3  ;
```

```
    r4 = r2 * r1;
    st (r10) = r4;
```

rr10 refers to rr9 of previous iteration (CORRECT!!!)
loop closing branch instruction (brc) rotates registers

Rotating register files avoid MVE !!!

Rotating register files

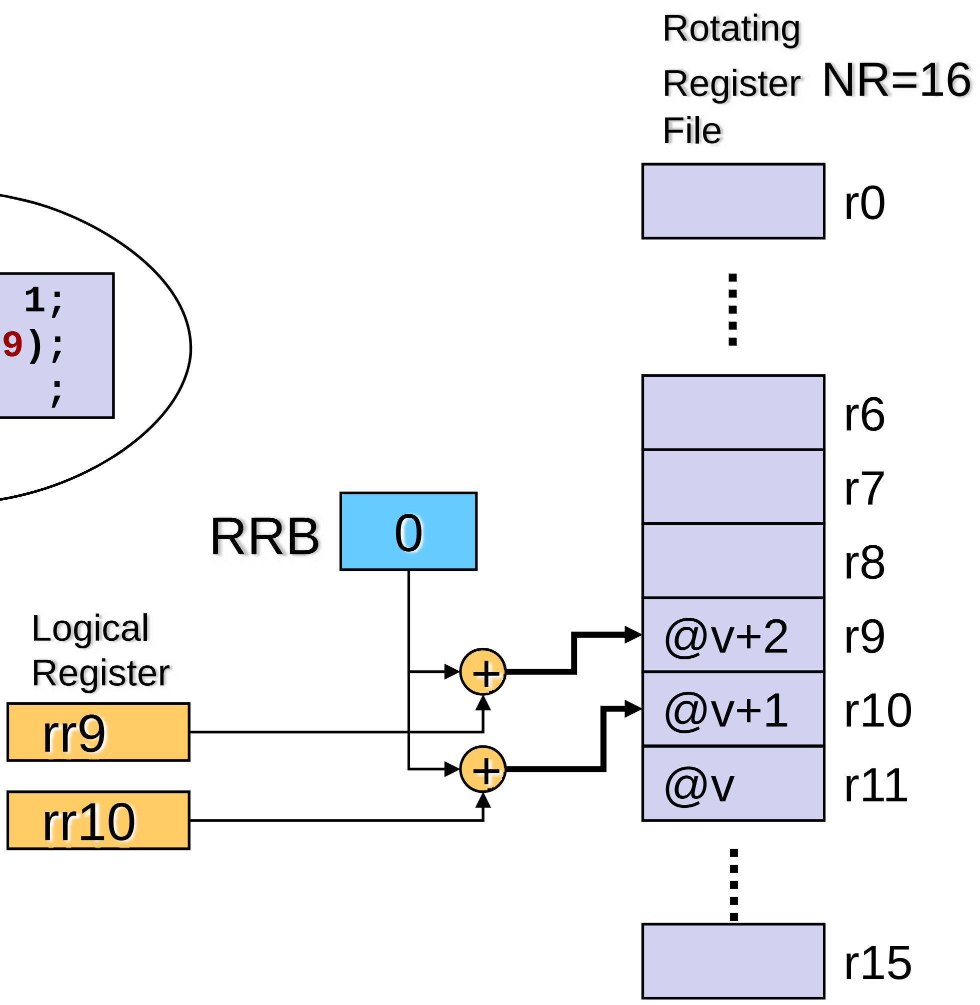


```
   rr11 = @V;
   r0 = N-1 ;
```

```
   rr10 = rr11 + 1;
   r1 = ld(rr10)  ;
   r2 = r1 + r3   ;
```

```
$1:r4 = r2 * r1 ; rr9 = rr10 + 1;
   st (rr10) = r4 ; r1 = ld(rr9);
   brc(r0) $1   ; r2 = r1 + r3   ;
```

```
   r4 = r2 * r1;
   st (r10) = r4;
```



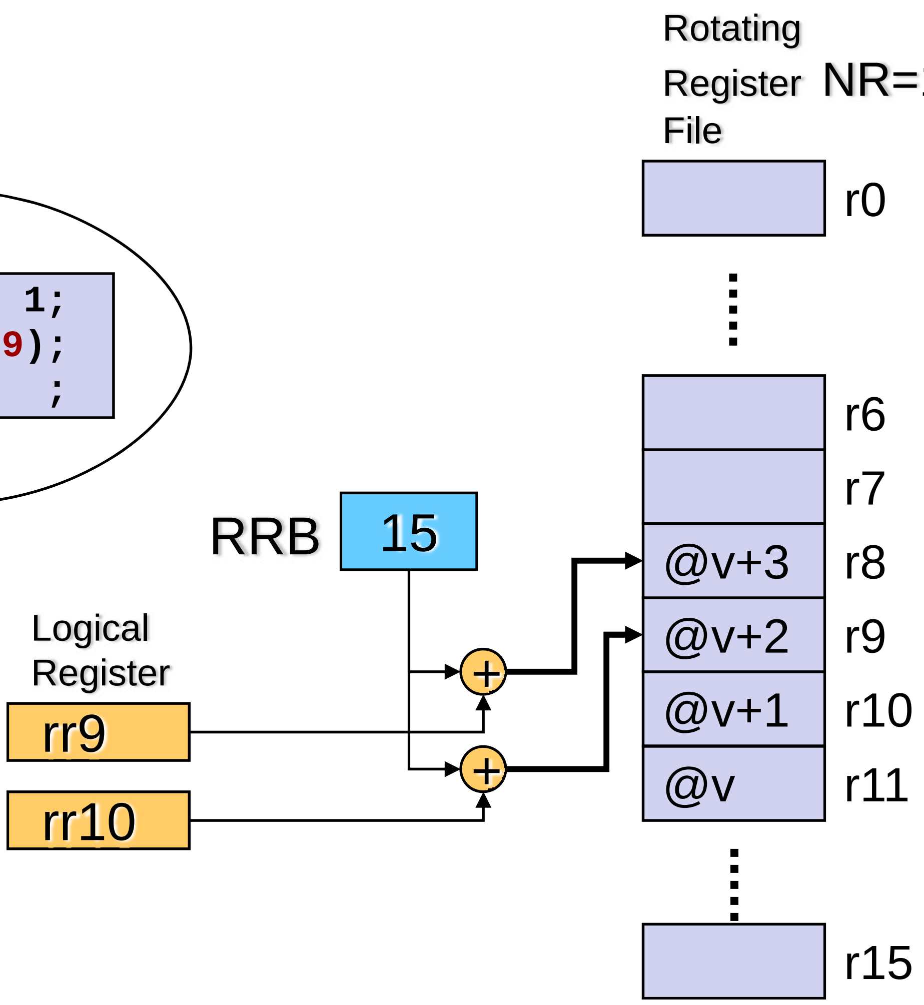
Rotating register files

```
    rr11 = @V;
    r0 = N-1 ;
```

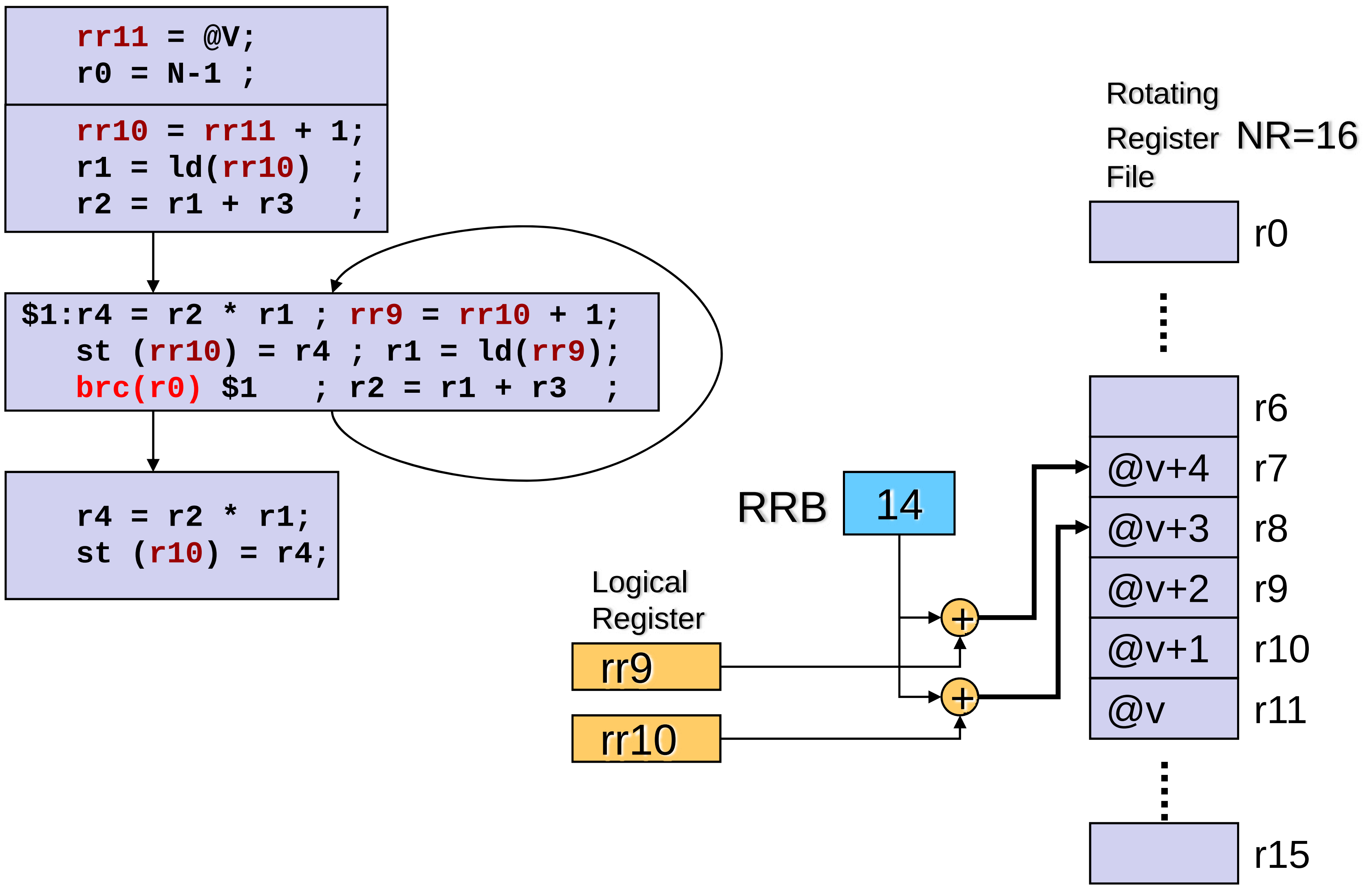
```
    rr10 = rr11 + 1;
    r1 = ld(rr10)  ;
    r2 = r1 + r3   ;
```

```
$1:r4 = r2 * r1 ; rr9 = rr10 + 1;
   st (rr10) = r4 ; r1 = ld(rr9);
   brc(r0) $1   ; r2 = r1 + r3   ;
```

```
    r4 = r2 * r1;
    st (r10) = r4;
```



Rotating register files



```
rr11 = @V;
r0 = N-1 ;
```

```
rr10 = rr11 + 1;
r1 = ld(rr10)  ;
r2 = r1 + r3   ;
```

```
$1:r4 = r2 * r1 ; rr9 = rr10 + 1;
   st (rr10) = r4 ; r1 = ld(rr9);
   brc(r0) $1   ; r2 = r1 + r3   ;
```

```
r4 = r2 * r1;
st (r10) = r4;
```

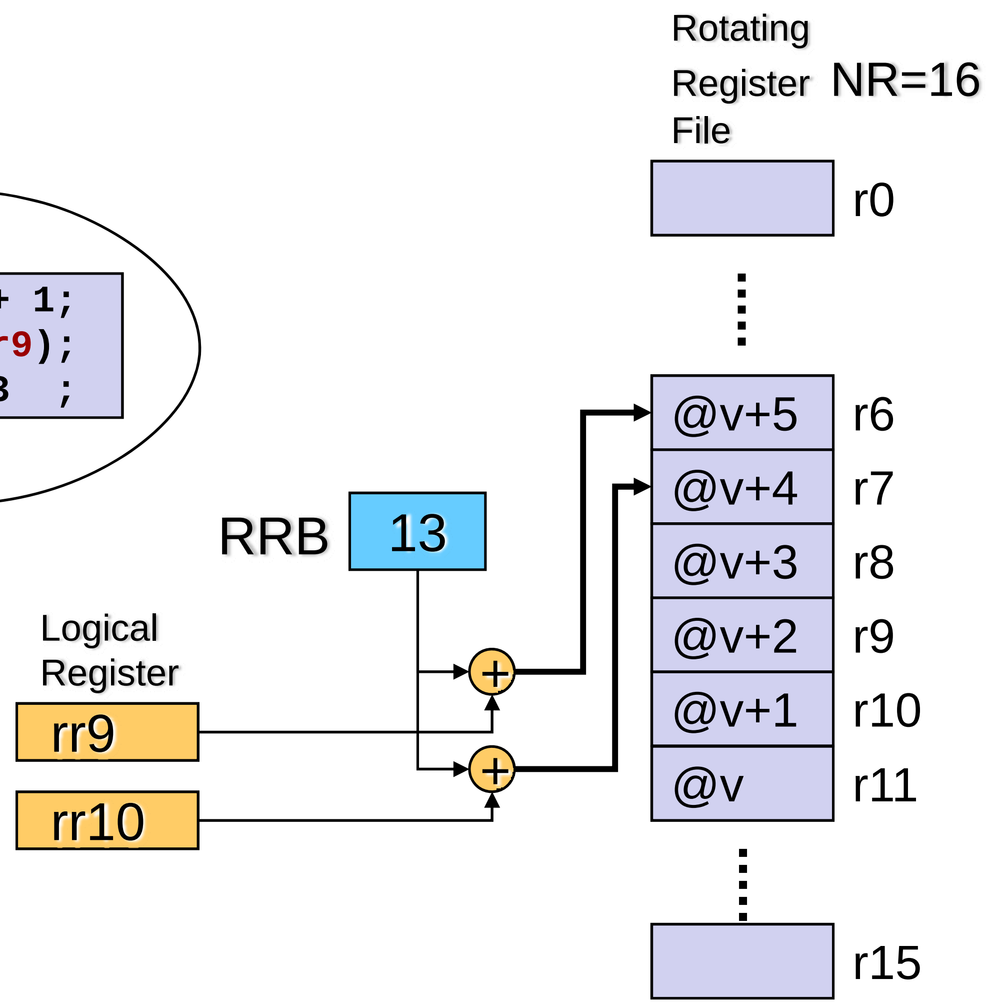
Rotating register files

```
    rr11 = @V;
    r0 = N-1 ;
```

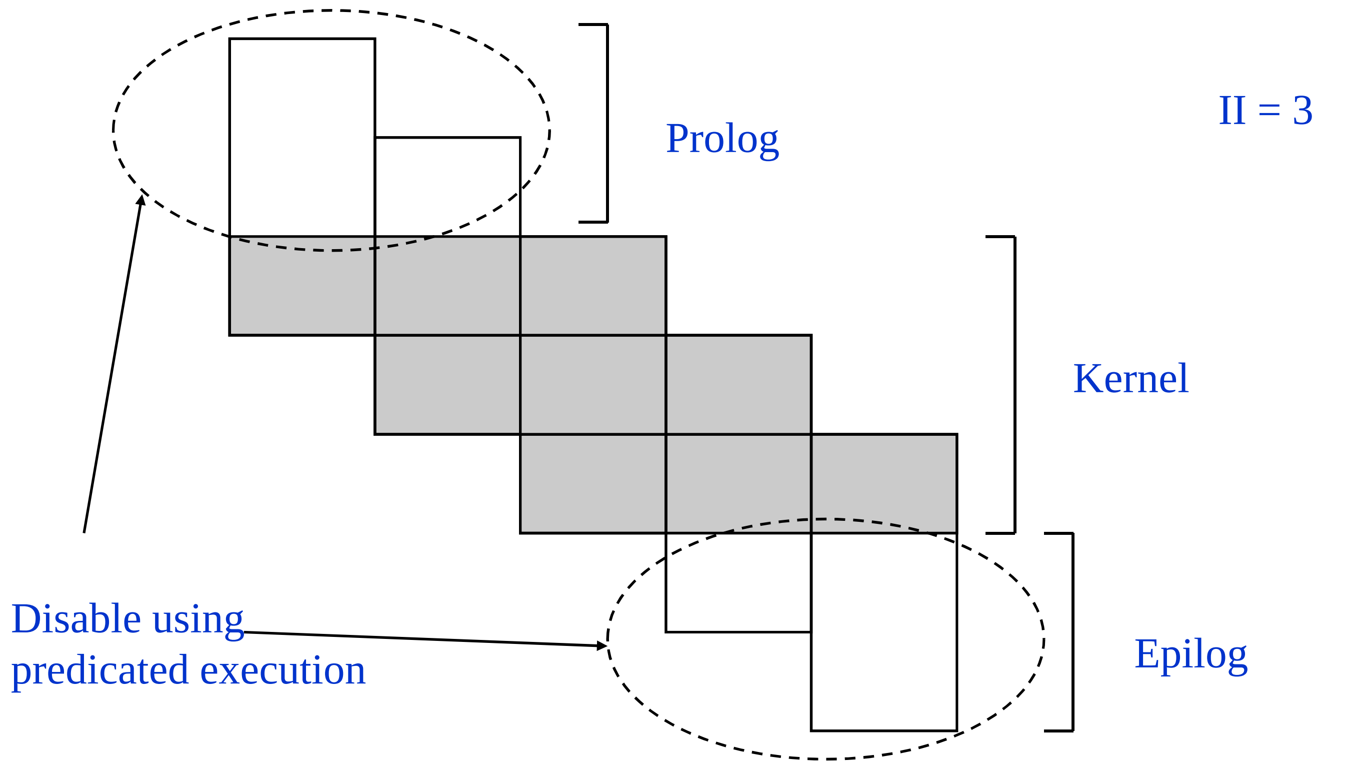
```
    rr10 = rr11 + 1;
    r1 = ld(rr10)  ;
    r2 = r1 + r3   ;
```

```
$1:r4 = r2 * r1 ; rr9 = rr10 + 1;
   st (rr10) = r4 ; r1 = ld(rr9);
   brc(r0) $1   ; r2 = r1 + r3  ;
```

```
    r4 = r2 * r1;
    st (r10) = r4;
```

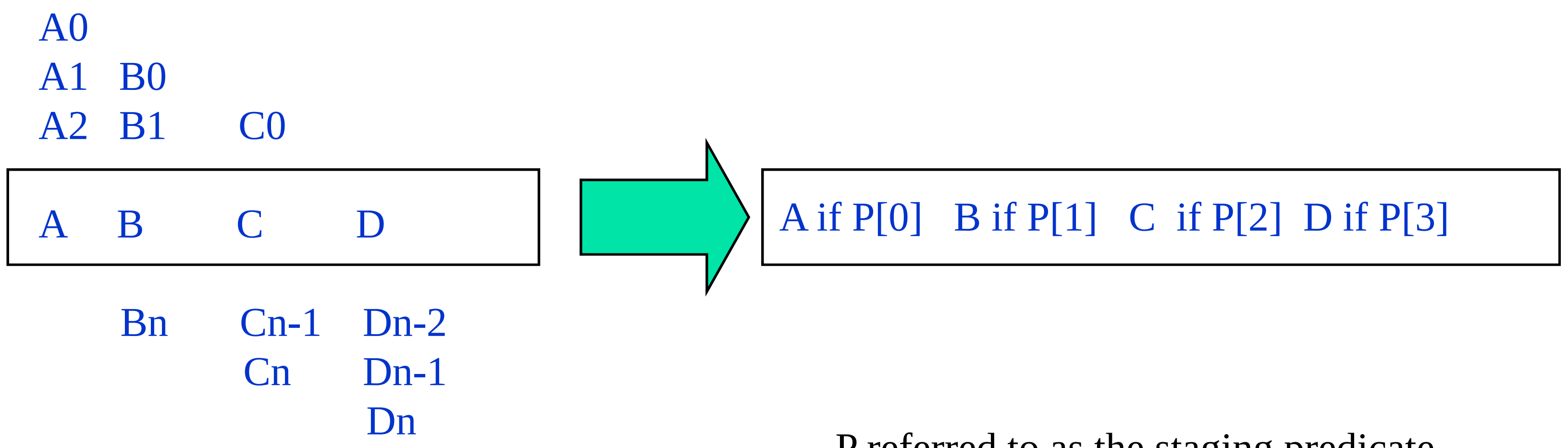


Removing Prolog/Epilog



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



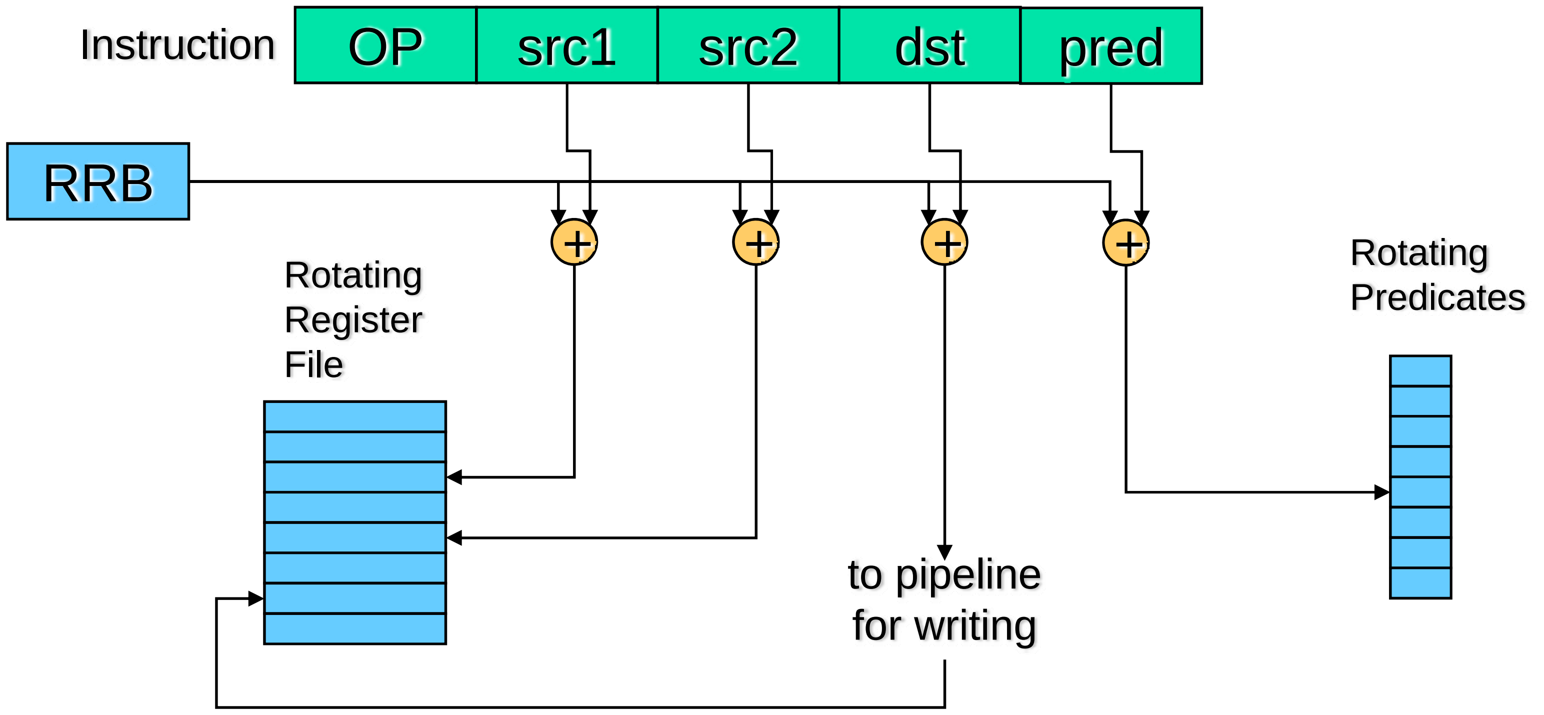
P referred to as the staging predicate

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

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

Rotating predicates

- Make predicates to rotate as well as registers
  - Contents of logical predicate Pi in iteration i
  - Are accessed through logical predicate Pi+1 in iteration i+1
- Uses same renaming hardware as rotating registers



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-1 in loop preheader
  - All rotating predicates are cleared
  - BRtop
    - 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 = 4, ESC = 3 /* Remember 0 relative!! */
Clear all rotating predicates
P[0] = 1

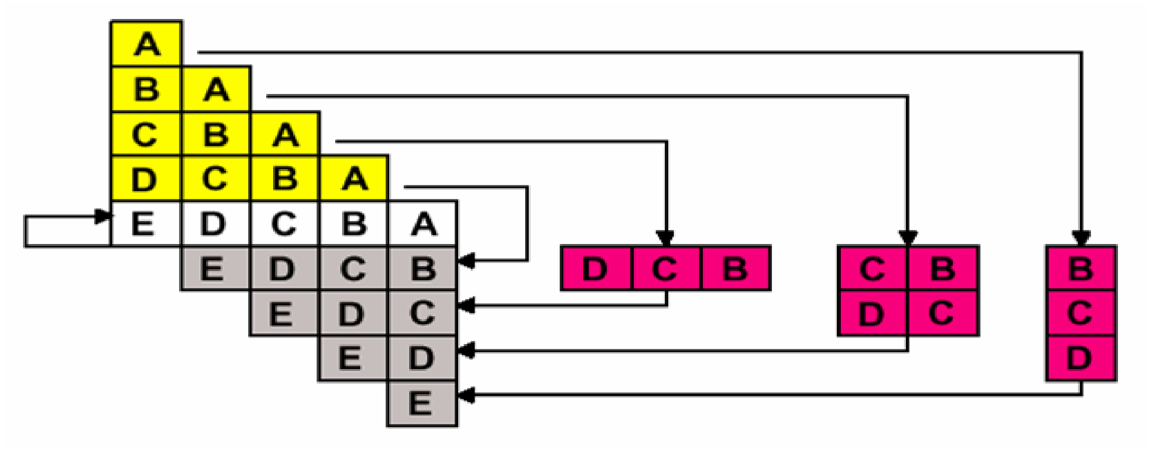
```
A if P[0];   B if P[1];   C if P[2];  D if P[3];  P[0] = BRtop;
```

LC	ESC	P[0]	P[1]	P[2]	P[3]				
4	3	1	0	0	0	A			
3	3	1	1	0	0	A	B		
2	3	1	1	1	0	A	B	C	
1	3	1	1	1	1	A	B	C	D
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

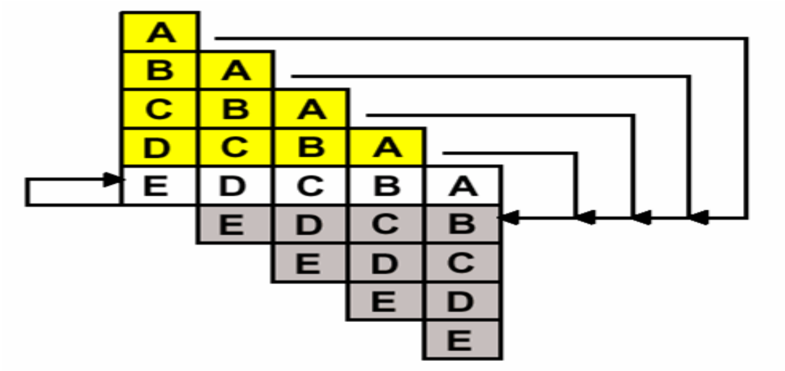
5 iterations, 4 stages, II = 1, Note 5 + 4 –1 iterations of kernel executed

Code schemas with Hardware Support

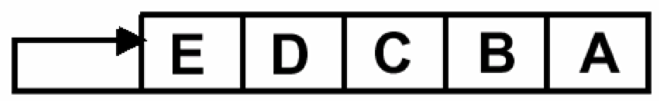
With Rotating Registers: we can avoid MVE



With Predicates and Rotating Registers : we can avoid mulitple epilogues



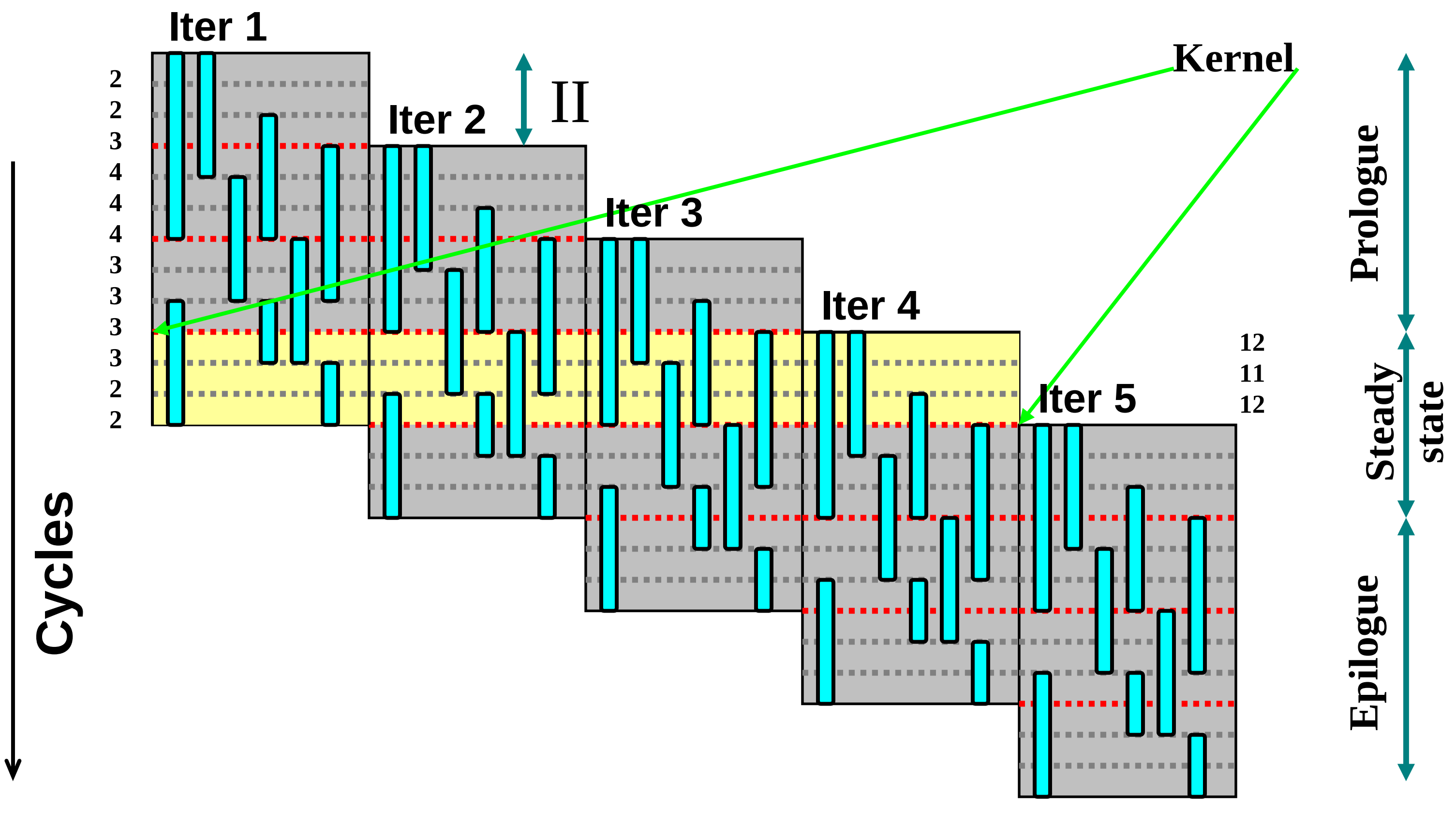
With Rotating Registers and Rotating Predicates: kernel-only code



From UIUC ECE 411 and NTU CA 718-Q lecture notes © 1999 Wen-mei W. Hwu

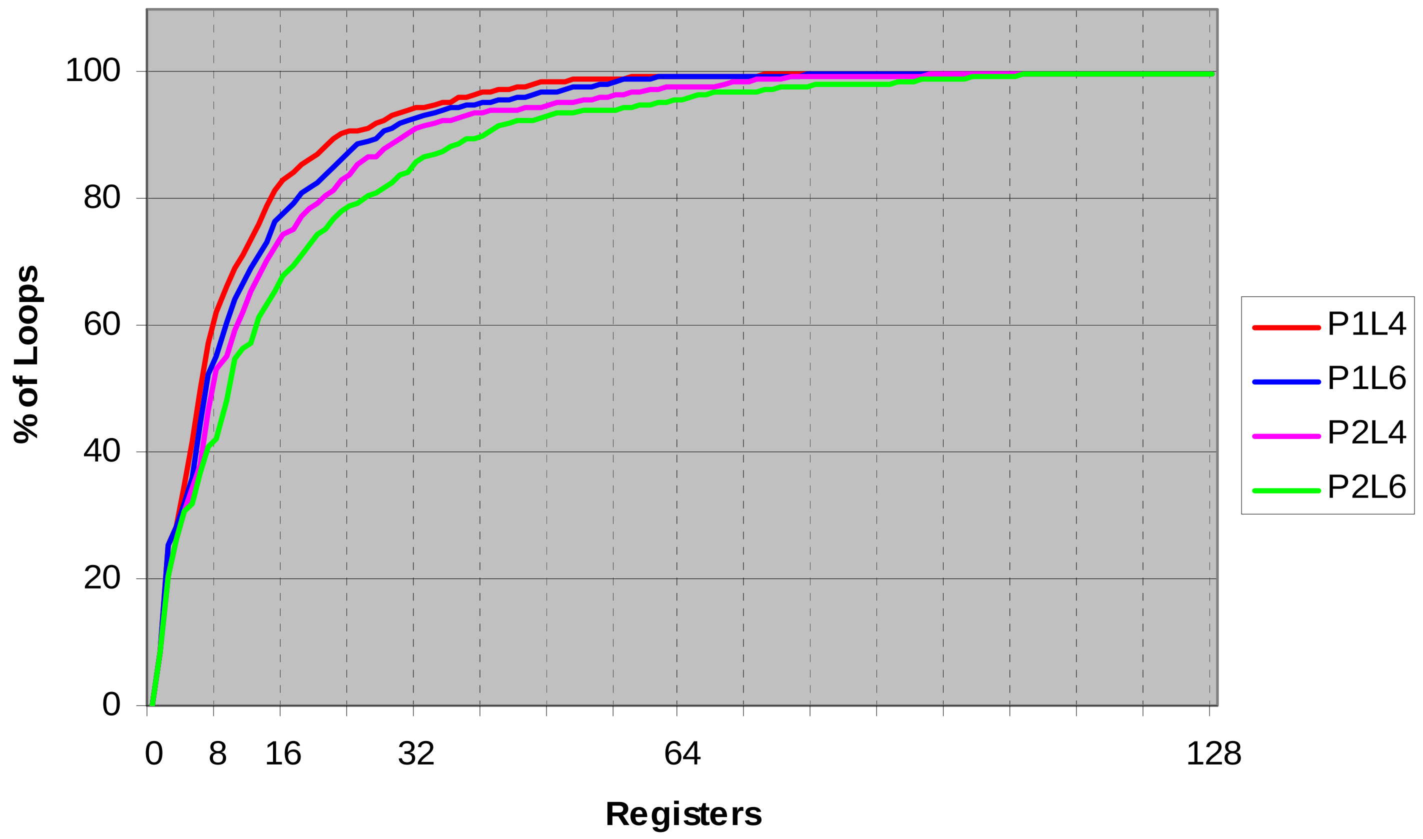
Register Requirements

- Aggressive scheduling techniques: high register requirements



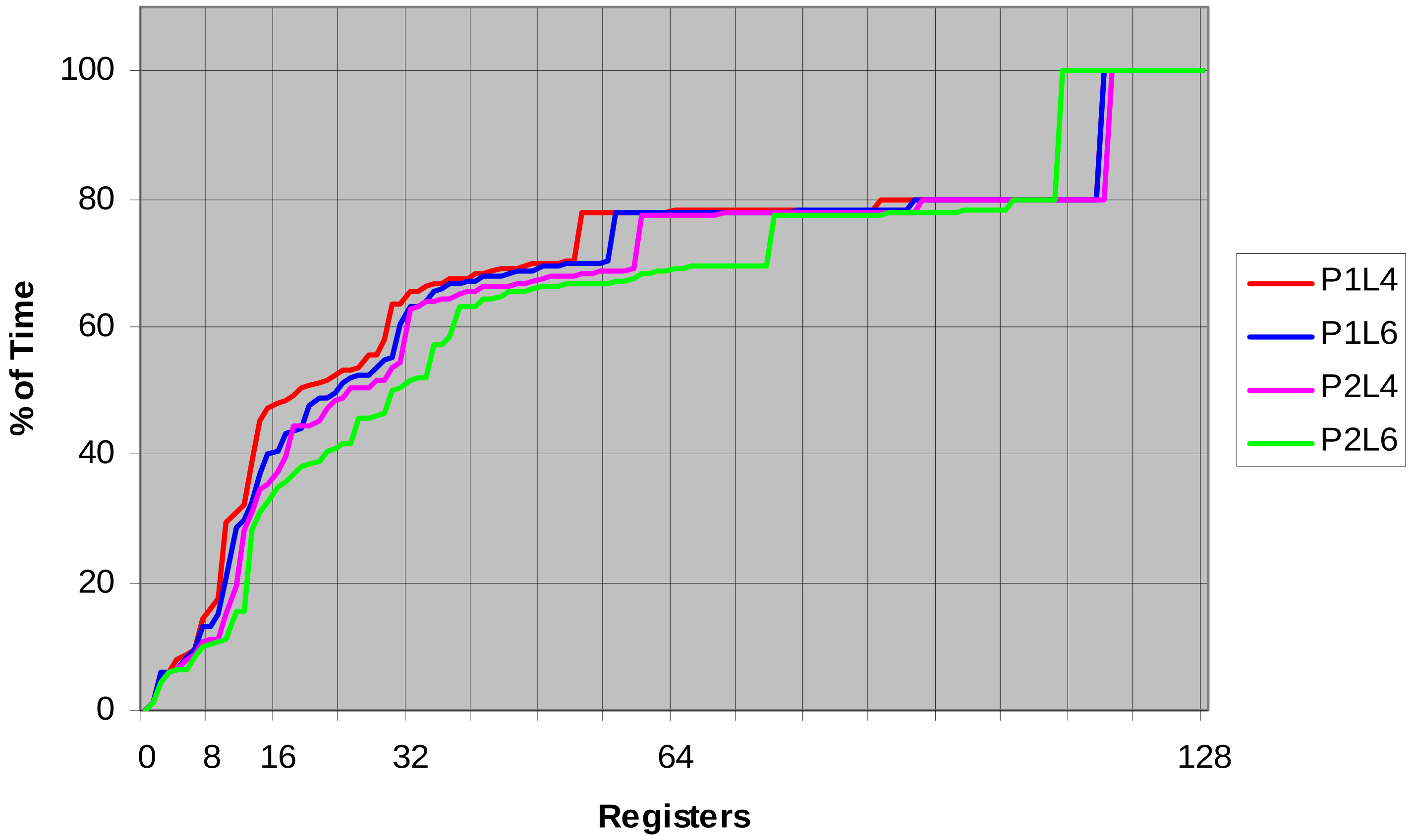
Register requirements

- Static register requirements



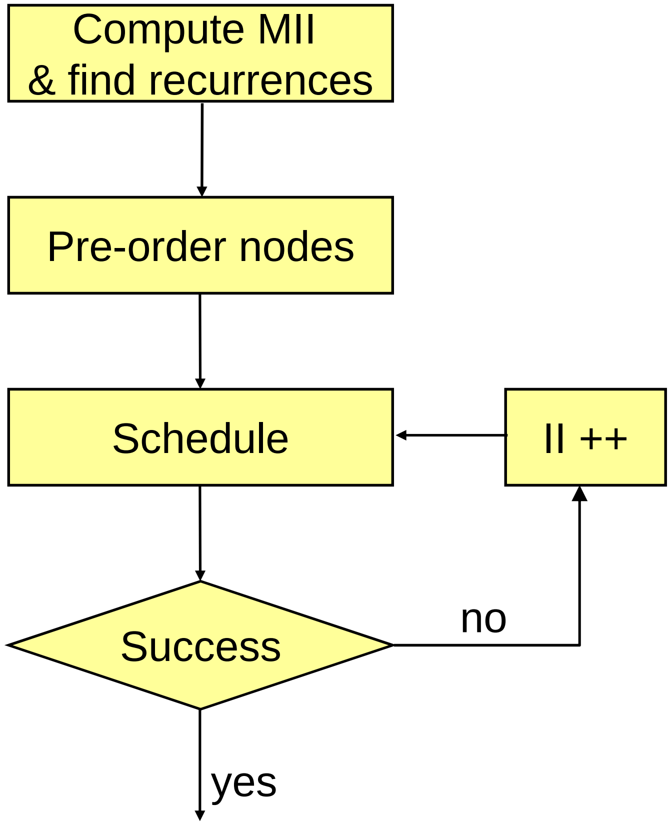
Register requirements

- Dynamic register requirements

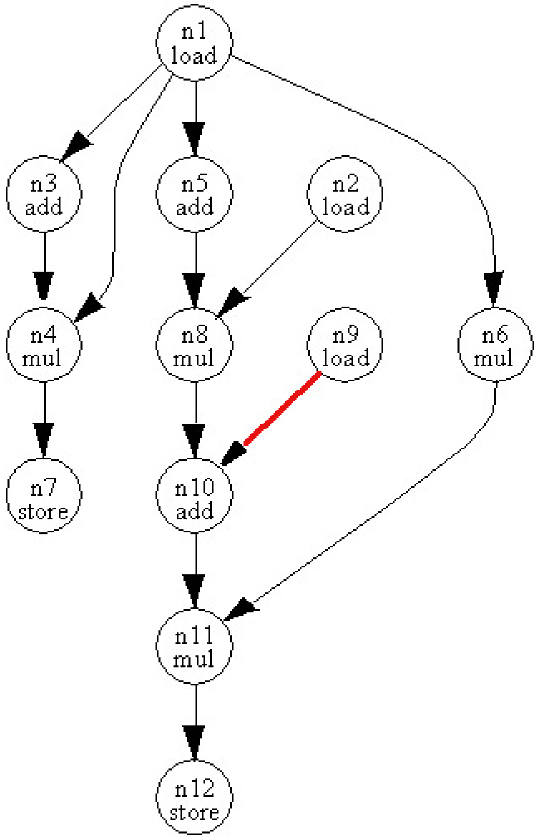


HRMS/SMS

- Proposed techniques:
  - HRMS: Hypernode Reduction Modulo Scheduling [micro95]
  - SMS: Swing Modulo Scheduling [pact96]
- Static priority function to pre-order nodes:
  - Hypernode reduction
    - Bidirectional ordering by adjacency
    - Special priority to recurrence circuits
  - Swing
    - Similar to above but faster and with enhanced critical path priority
- Simple scheduling:
  - bidirectional greedy modulo scheduling
  - Nodes with only predecessors are scheduled top-down
  - Nodes with only successors are scheduled bottom-up



Top-down vs. HRMS



Hardware configuration:

1 add unit

1 mul unit

2 load/store units

Latencies:

add: 2 cycles

mul: 2 cycles

load: 2 cycles

store: 1 cycle

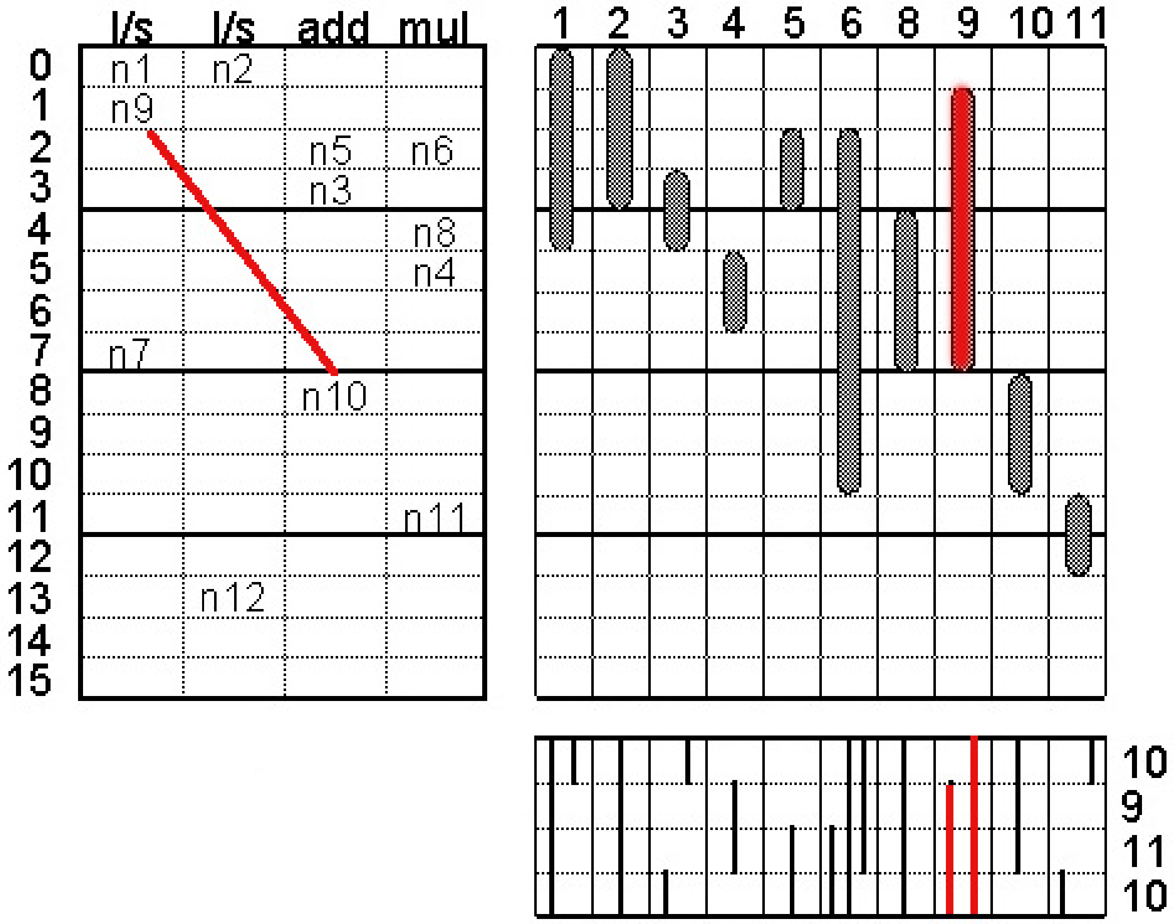
Pre-ordering step:

Top-down={ 1, 2, **9**, 3, 5, 6, 4, 8, 7, 10, 11, 12 }

HRMS={ 1, 3, 5, 6, 4, 7, 8, 10, 11, **9**, 2, 12 }

SMS={ 12, 11, 10, 8, 5, 6, 1, 2, **9**, 3, 4, 7 }

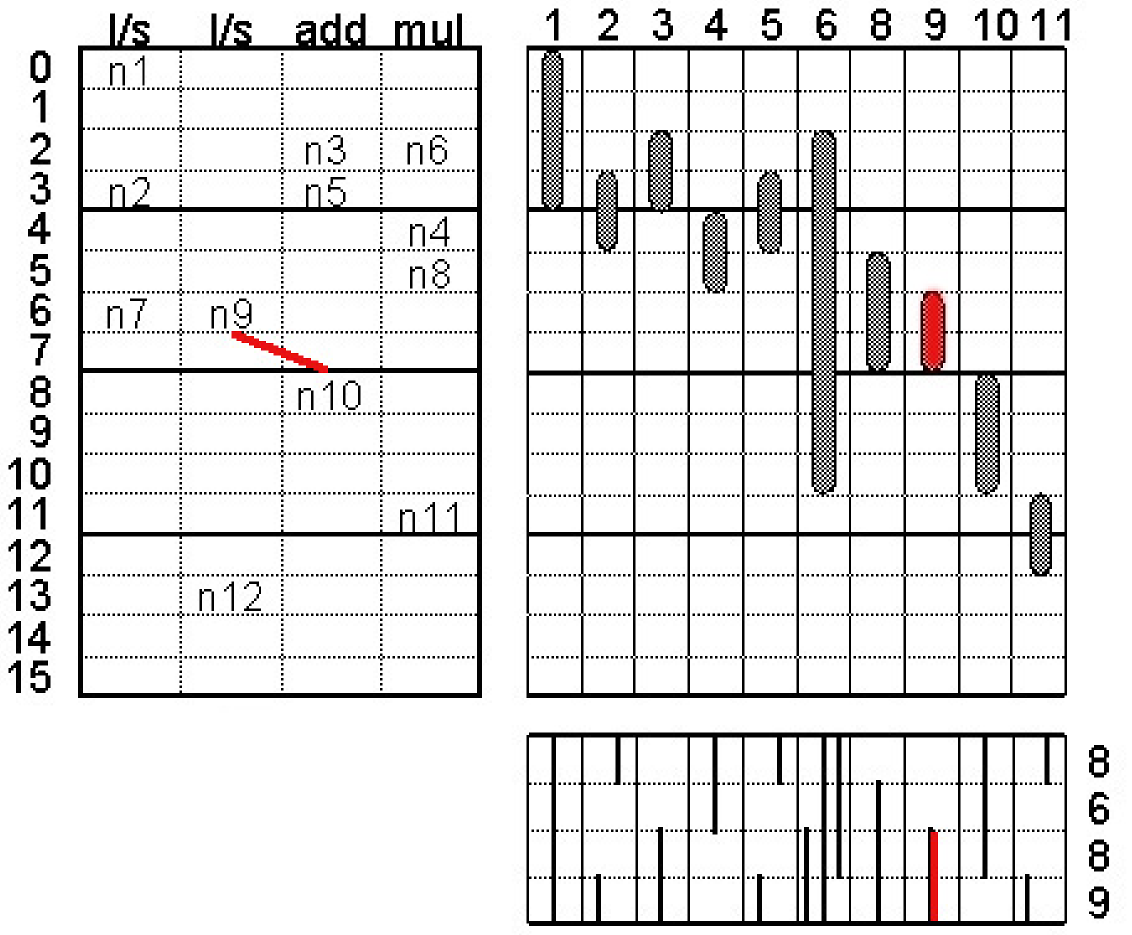
Top-down vs. HRMS



Pre-ordering step:

Top-down={ 1, 2, 5, 8, **9**, 3, 10, 6, 4, 11, 12, 7 }

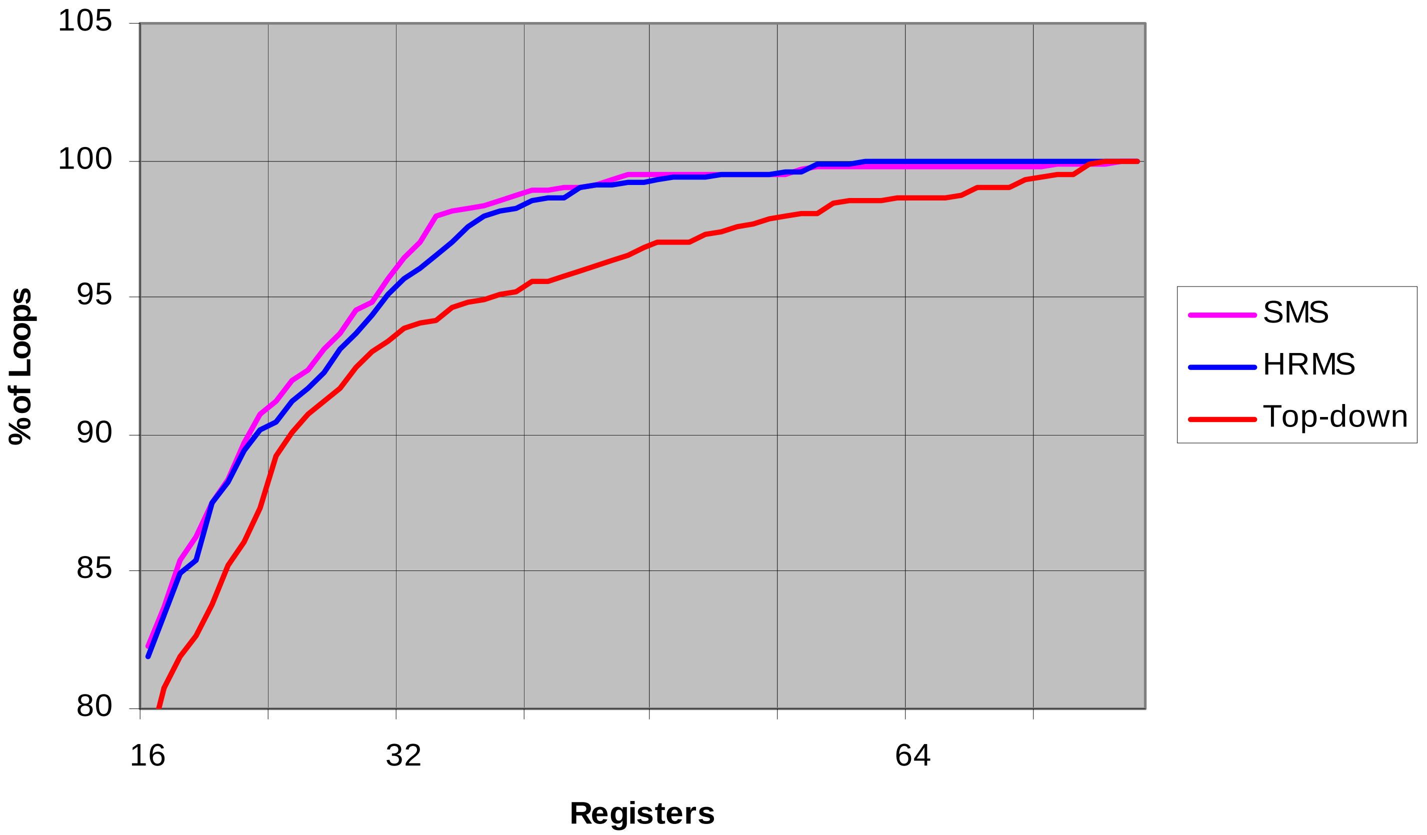
Top-down vs. HRMS



Pre-ordering step:

HRMS={ 1, 3, 5, 6, 4, 7, 8, 10, 11, 9, 2, 12 }

HRMS and SMS register requirements


% of Loops
Registers
SMS
HRMS
Top-down