CE 658
Advanced Computer Architecture
Fall 2018

Compile-time ILP extraction Modulo Scheduling

Nikos Bellas

Computer and Communications Engineering Department
University of Thessaly

Readings for this lecture

- **Hardware and Software for VLIW and EPIC**
  - H&P, Appendix G in v.4
  - H&P, Chapter 4 in v.3
- **Scheduling techniques**
  - B. Rau, et. al *"Code Generation Schema for Modulo Scheduled Loops"*, Micro 24, December 1992
  - CT Hwuang, et. al *"PLS: A Scheduler for Pipeline Synthesis", IEEE Transactions on CAD,* September 1993
  - Josep Llosa, et. al. *"Swing Modulo Scheduling: A Life-Time Sensitive Approach,"* , PACT 1996

Static scheduling – Explicit loop unrolling

```
L:  fld   f0,0($1)
    fadd  f4,f0,f2
    fst   f4,0($1)
    addi  $1,$1,-8
    bne   $1,$2,L
```

Schedule code for a superscalar processor considering the following latencies:

- Memory instr: 2 cycles
- FP instr: 3 cycles
- Branch : 1 cycle

Static scheduling – Explicit loop unrolling

Mem 1	Mem 2	FP 1	FP 2	Int+Br
fld f0,0($1)				
		fadd f4,f0,f2		
fst f4,0($1)				
				addi $1,$1,-8
				beq $1,$2,L

First attempt: straightforward scheduling
5 instructions in 9 cycles (IPC = 0.56)
11% scheduling efficiency (5 instructions in 9*5=45 slots)
Only 2 FP registers used ($f0, $f4)

Static scheduling – Explicit loop unrolling

Schedule that eliminates all stalls

Mem 1	Mem 2	FP 1	FP 2	Int+Br
fld f0,0($1)	fld f6,-8($1)			
fld f10,-16($1)	fld f14,-24($1)			
fld f18,-32($1)	fld f22,-40($1)	fadd f4,f0,f2	fadd f8,f6,f2	
fld f26,-48($1)		fadd f12,f10,f2	fadd f16,f14,f2	
		fadd f20,f18,f2	fadd f24,f22,f2	
fst f4,0($1)	fst f8,-8($1)	fadd f28,f26,f2		
fst f12,-16($1)	fst f16,-24($1)			addi $1,$1,-56
fst f20,-32($1)	fst f24,-40($1)			
fst f28,-48($1)				beq $1,$2,L

Much better with implicit loop unrolling (7 times) and scheduling (software pipelining)
23 instructions in 9 cycles (IPC = 2.56)
51% scheduling efficiency
14 FP registers used (register pressure)

Dynamic vs Static Scheduling

- Superscalar processors decide on-the-fly how many and which instructions to issue
- Complex hardware is used to check for dependencies and schedule operations
- The window of instructions that are checked each time is limited by the hardware resources
  - All checking has to be done at real time
  - Minimal compiler assistance required

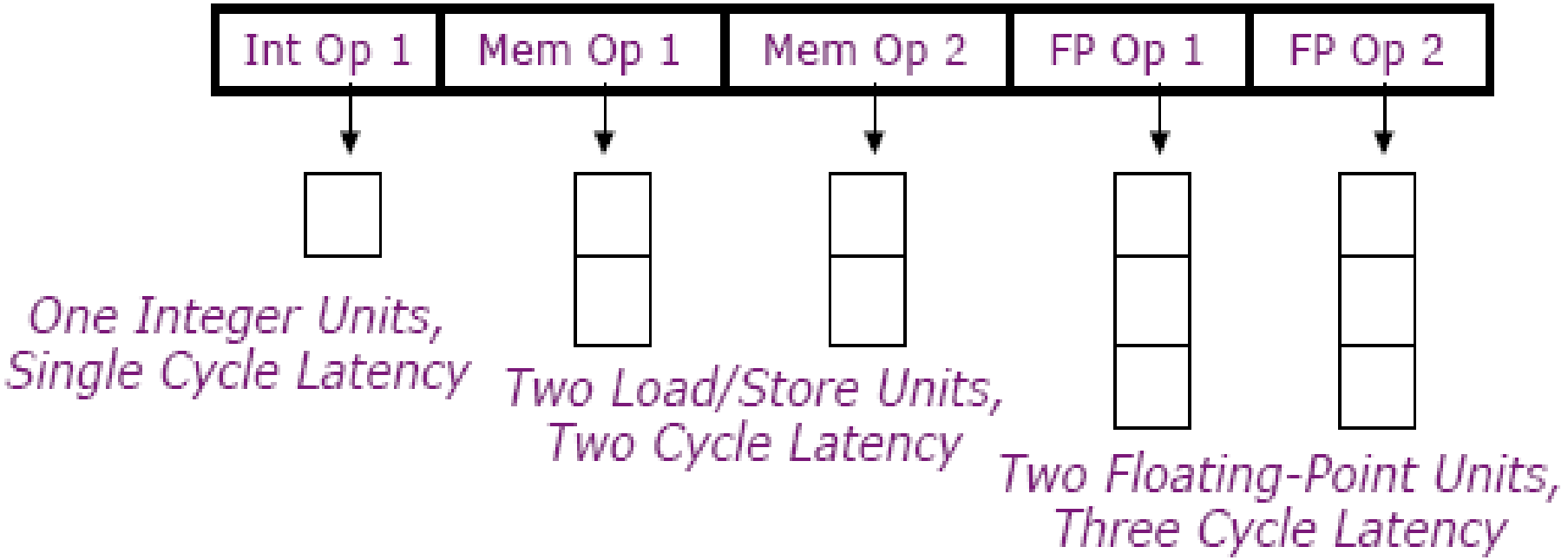
Dynamic vs Static Scheduling

- Statically scheduled processors rely on the compiler to achieve good performance
- Main objective of the compiler is to reduce potential data hazards
- The processor still has to stall in case of hazards
- The extreme case of static scheduling is VLIW

VLIW technology

- Due to the growing complexity to scale the dynamic scheduling techniques, the industry started to re-examine explicit encoding of multiple operations (mid 90's)
- Rely on compiler technology to form the instructions in an issue packet and schedule all of them simultaneously
  - VLIW – Very Long Instruction Word
  - Multi-operation instructions (64-128 bits or more)
- The hardware does not check for dependencies
- no dependencies among operations in the same issue word

VLIW approach



- Only sensible for wide-issue processors (> 4 slots)
- Multiple operations packed in one instruction
- Each operation slot corresponds to a specific FU
- Constant operation latencies specified
- Parallelism within an instruction.
- Basic compiler techniques are the same for VLIW or statically scheduled processors

VLIW approach

- The compiler:
  - schedules for maximum throughput
    - As many useful operations as possible within an instruction
    - With the help of profiling feedback
    - Modulo scheduling, trace scheduling, speculation, predication, superblocks, hyperblocks .....
  - Guarantees intra-instruction parallelism
  - Schedules to avoid data hazards
    - pure VLIW engines have no interlocks !

VLIW approach

- VLIW (at least the original forms) has several short-comings that precluded it from becoming mainstream:
  - VLIW instruction sets are not backward compatible between implementations. When wider implementations (more execution units) are built, the instruction set for the wider machines is not backward compatible with older, narrower implementations.
  - Load responses from a memory hierarchy which includes CPU caches and DRAM do not have a deterministic delay. This makes static scheduling of load instructions by the compiler very difficult and inefficient.
  - Code explosion with all these compiler techniques

Josh Fisher. Very Long Instruction Word architectures and the ELI-512 . *Proceedings of the 10th annual international symposium on Computer architecture*. 1983. New York, NY

This work resulted in a company called Multiflow.

Modulo Scheduling

A Class of Methods for Software Pipelining Loops

Software Pipelining

- A technique for scheduling instructions to exploit ILP in inner loops
  - Many programs spend most of their time there
  - Parallelism between loop iterations
- The technique is based on overlapping multiple loop iterations and executing them in parallel
  - Similar to multiple instruction execution in hardware pipelining
- It attempts to achieve the benefits of loop unrolling without performing explicit loop unrolling

Modulo scheduling

- *Modulo scheduling* is an algorithm for software pipelining
  - All loop iterations have a common schedule
  - Modulo scheduling requires hardware support
- The number of cycles between scheduling successive loop iterations is called iteration interval (II)
  - We want to minimize II
  - The most important metric to evaluate a SW pipelining algorithm

Let us start with an example

Matrix-Vector multiply

Source Code

```
initialize C to 0.0
for (j=0; j<m; j++) {
        for (i=0; i<n; i++) {
                C[i] = C[i] + A[j] * B[j][i]
        }
}
```

Assembly Code for Inner Loop

Inst.	Assembly
1	L1: f1 = MEM(r8+r4)
2	f5 = MEM(r2+r4)
3	f6 = f3 * f5
4	f1 = f1 + f6
5	r4 = r4 + 4
6	bgtz --r5, L1

Register Contents

f1 = C[i]
f3 = A[j]
f5 = B[j][i]
r8 = &C[0]
r2 = &B[j][0]
r4 = 4*i
r5 = n

Case Study: Scheduling

- Processor model
  - 6-issue, fully-pipelined, 2 cycle latency
  - 3 integer/load/store units
  - 2 floating-point units
  - 1 branch per cycle
- Schedule for a single iteration
  - 5 cycles per iteration

Loop Unrolling and Iteration folding

```
Inst.  Assembly
1      L1:  f1 = MEM(r8+r4)
2           f5 = MEM(r2+r4)
3           f6 = f3 * f5
4           f1 = f1 + f6
5           r4 = r4 + 4
6           bgt ((--r5) 0)  L1
```

	M1	M2	M3	FP1	FP2	B
0	1_1	2_1	5_1			
1	1_2	2_2	5_2			
2				3_1		
3				3_2		
4					4_1	6_1
5					4_2	6_2

	M1	M2	M3	FP1	FP2	B	
0	1_1	2_1	5_1				Prologue
1	1_2	2_2	5_2				
2	1_3	2_3	5_3	3_1			
3	1_4	2_4	5_4	3_2			
4	1_5	2_5	5_5	3_3	4_1	6_1	Steady State
5	1_6	2_6	5_6	3_4	4_2	6_2	
6	1_7	2_7	5_7	3_5	4_3	6_3	
7	1_8	2_8	5_8	3_6	4_4	6_4	
8				3_7	4_5	6_5	Epilogue
9				3_8	4_6	6_6	
10					4_7	6_7	
11					4_8	6_8	

1 iteration in 5 cycles
Throughput = 0.20 results/cycle
2 iterations in 6 cycles
Throughput = 0.33 results/cycle

8 iterations in 12 cycles
Throughput = 0.67 results/cycle
N iterations in N+4 cycles

Loop Unrolling and Iteration folding

	M1	M2	M3	FP1	FP2	B
0	1_1	2_1	5_1			
1	1_2	2_2	5_2			
2	1_3	2_3	5_3	3_1		
3	1_4	2_4	5_4	3_2		
4	1_5	2_5	5_5	3_3	4_1	6_1
5	1_6	2_6	5_6	3_4	4_2	6_2
6	1_7	2_7	5_7	3_5	4_3	6_3
7	1_8	2_8	5_8	3_6	4_4	6_4
8				3_7	4_5	6_5
9				3_8	4_6	6_6
10					4_7	6_7
11					4_8	6_8

Prologue

Steady State

Epilogue

8 iterations in 12 cycles
Throughput = 0.67 results/cycle

0	1_1	2_1	5_1			
1	1_2	2_2	5_2			
2	1_3	2_3	5_3	3_1		
3	1_4	2_4	5_4	3_2		
n = 5,6,7,8	1_n	2_n	5_n	3_{n-2}	4_{n-4}	6_{n-4}
8				3_7	4_5	6_5
9				3_8	4_6	6_6
10					4_7	6_7
11					4_8	6_8

Fold the steady state

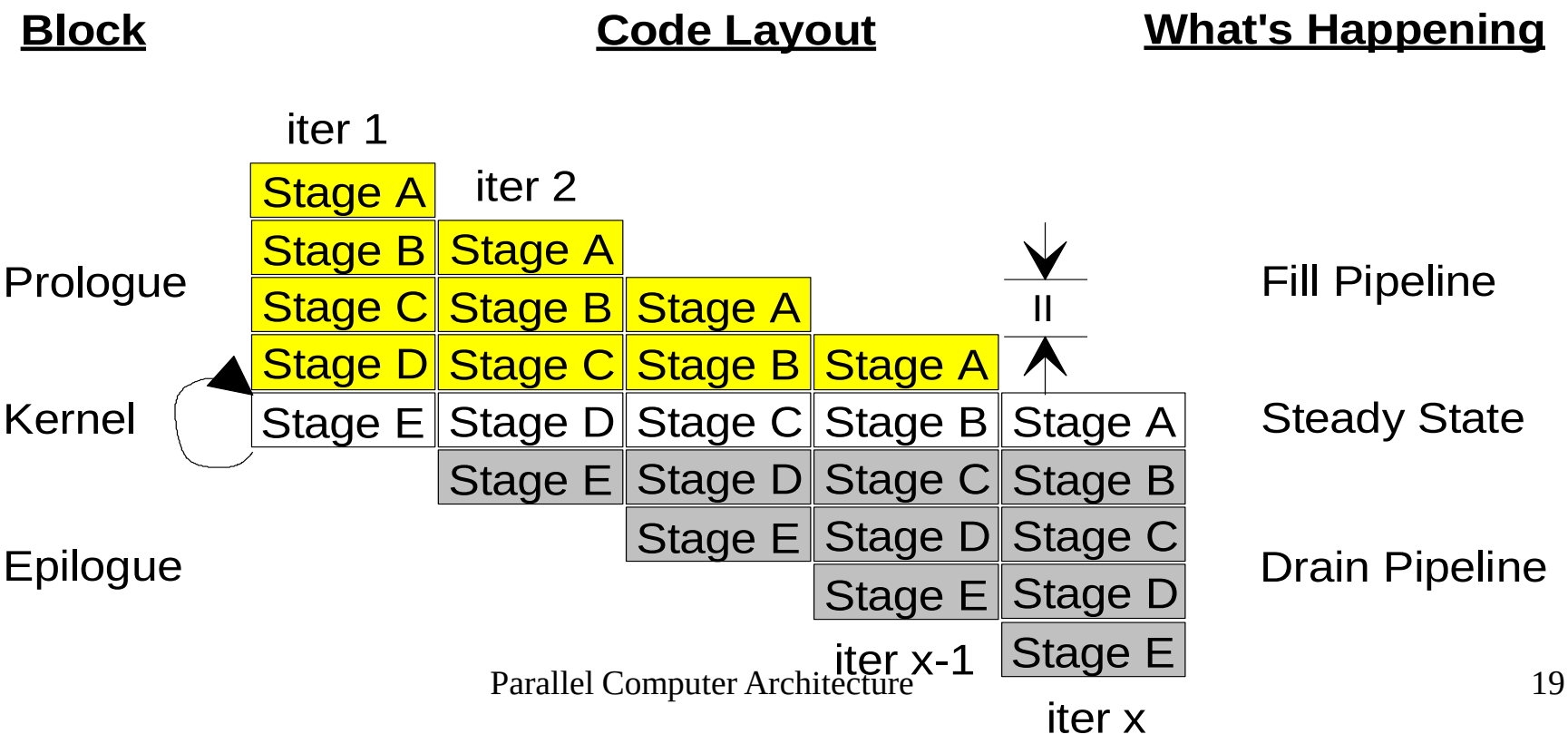
1_n	2_n	5_n	3_{n-2}	4_{n-4}	6_{n-4}

Only the kernel

- If we unroll enough loops, we can improve to 1 result/cycle
- Latency for first iteration is 5 cycles
- After that one iteration per each cycle
- Iteration Interval = 1

Software Pipelining

- Each *stage* consists of II cycles
- The number of stages in one iteration is called *stage count (SC).* Here SC=5



Modulo Scheduling

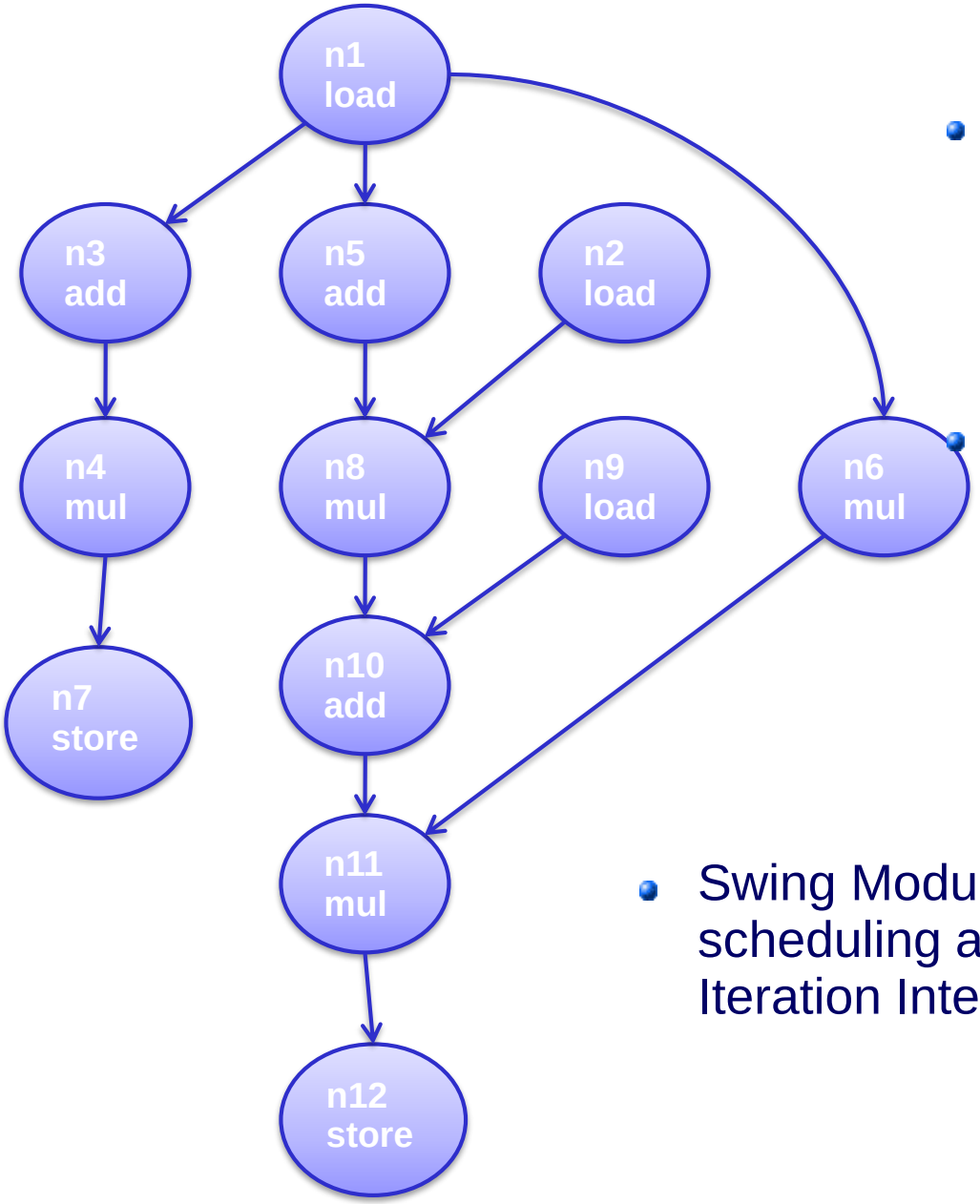
- Simplifies the process of software pipelining by:
  - using a common schedule for all iterations
  - initiating iterations at a constant rate
    - Initiation Interval (II)

Swing Modulo Scheduling (SMS)

A Modulo Scheduling Algorithm

- Four stages for developing a schedule
  - Construction of the Control Data Flow Graph (CDFG) of the program
  - Computation of minimum iteration interval (II)
  - Ordering of nodes (in what order we schedule the nodes)
  - Scheduling of ordered nodes (mod II)

Swing Modulo Scheduling Example



- Processor model
  - 2 integer/load/store units
  - 1 add unit
  - 1 mul unit
- Latencies
  - add: 2 cycles
  - mul: 2 cycles
  - load: 2 cycles
  - store: 1 cycle

- Swing Modulo Scheduling (SMS) is a modulo scheduling algorithm that tries to minimize a) the Iteration Interval (II), and b) register requirements

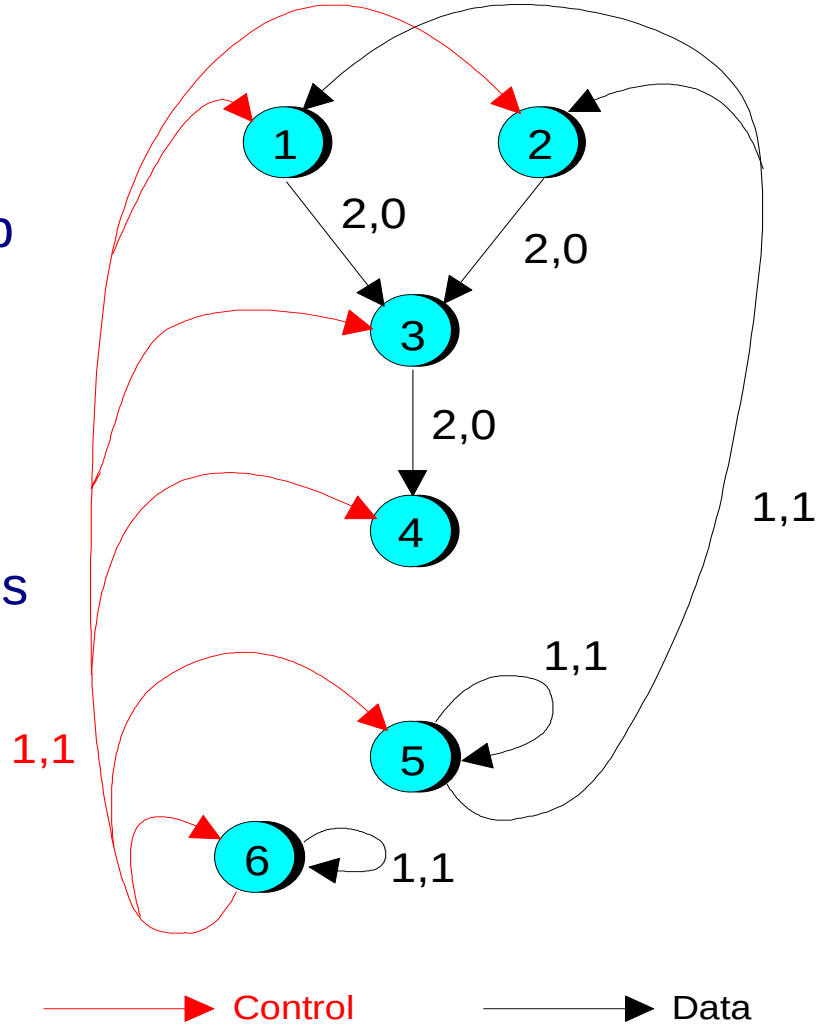
Control Data Flow Graph (CDFG)

Dependence graph G(U,E)

- U is the set of instructions in the loop
- E is the set of edges in the loop that show data flow and control flow

Each edge is annotated with two numbers *(λ, δ)*.
The latency *λ* shows how many cycles to produce a result
The distance *δ* is the number of loop iterations that separate the two instructions.

Dependence Graph for Inner Loop



Initiation Interval

- Determine lower bound on II (MII)
- Attempt to generate a schedule at that II
- There are two constraints on the MII
  - Resource constraints
  - Recurrence constraints

Resource Constrained MII

- Let M_k the number available functional units of type *k.*
- Let N_k be the number of operations of an iteration which are executed in a functional unit of type *k*.
- Lower bound of latency due to type *k* FU is $\left\lceil N_k / M_k \right\rceil$
- For example, if a DFG contains 5 additions and 6 subtractions, the lower bound on latency given 3 ALUs is $\left\lceil (5+6)/3 \right\rceil = 4$
- Lower bound of II of the DFG is $\mathrm{ResII} = \max_{1 \le k \le m} \left\lceil N_k / M_k \right\rceil$

Resource constrained MII: example

Assembly Code Loop

```
L1: f3 = MEM(r8+r4)
    f5 = MEM(r2+r4)
    f6 = f3 * f5
    MEM(r9+r4) = f6
    r4 = r4 +4
    ble (r4 r7) L1
```

ResMII = max(ResMII of each resource)
ResMII = max(ceil(4/3), 1, 1) = 2

Therefore, II cannot be smaller than 2

M1 = 3 integer/load/store units

M2 = 1 floating-point units

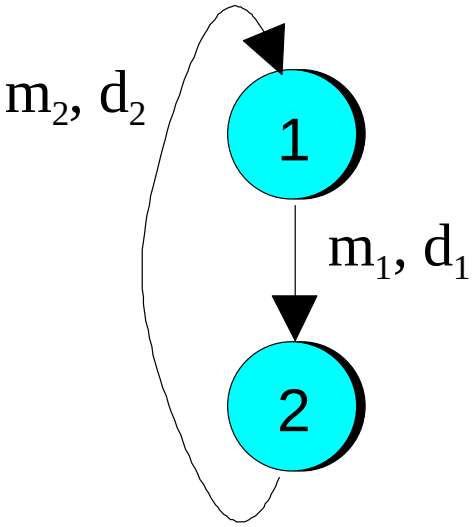
M3 = 1 branch per cycle

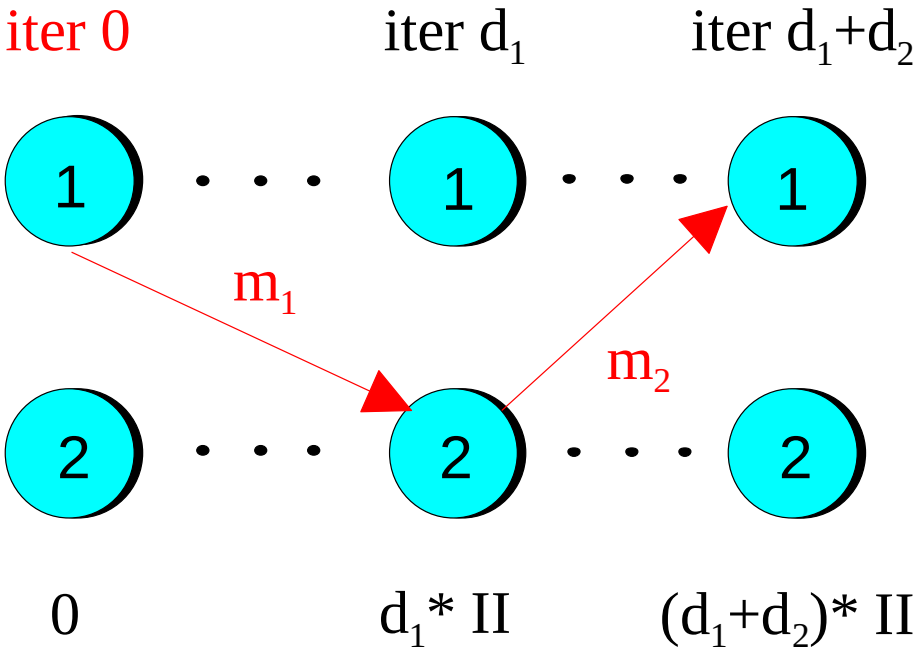
n1 = 4 integer/memory operations

n2 = 1 FP operation

n3 = 1 branch operation

Recurrence-constrained MII





Delay(c) = $m_1 + m_2$

Distance(c) = $d_1 + d_2$

Delay(c) $\leq$ Distance(c) * II

Cross-iteration Dependencies
e.g. A[i] = A[i-1] + 1

Recurrence Constraints

- Suppose there is a dependency cycle in the CDFG between operations:

$o_1 @ 1 \rightarrow o_2 @ 1 \rightarrow .. \rightarrow o_c @ 1 \rightarrow o_1 @ 2$

  - The notation $o_i @ n$ means : operation o_i at iteration n

- Let $M_i = \sum_{1 \le i \le c} m_i$ be the delay of all operations in the loop
- Let be the sum of the distances in the loop

$$D_i = \sum_{1 \le i \le c} d_i$$

- Lower bound of the II due to this loop is $\left\lceil \frac{M_i}{D_i} \right\rceil$

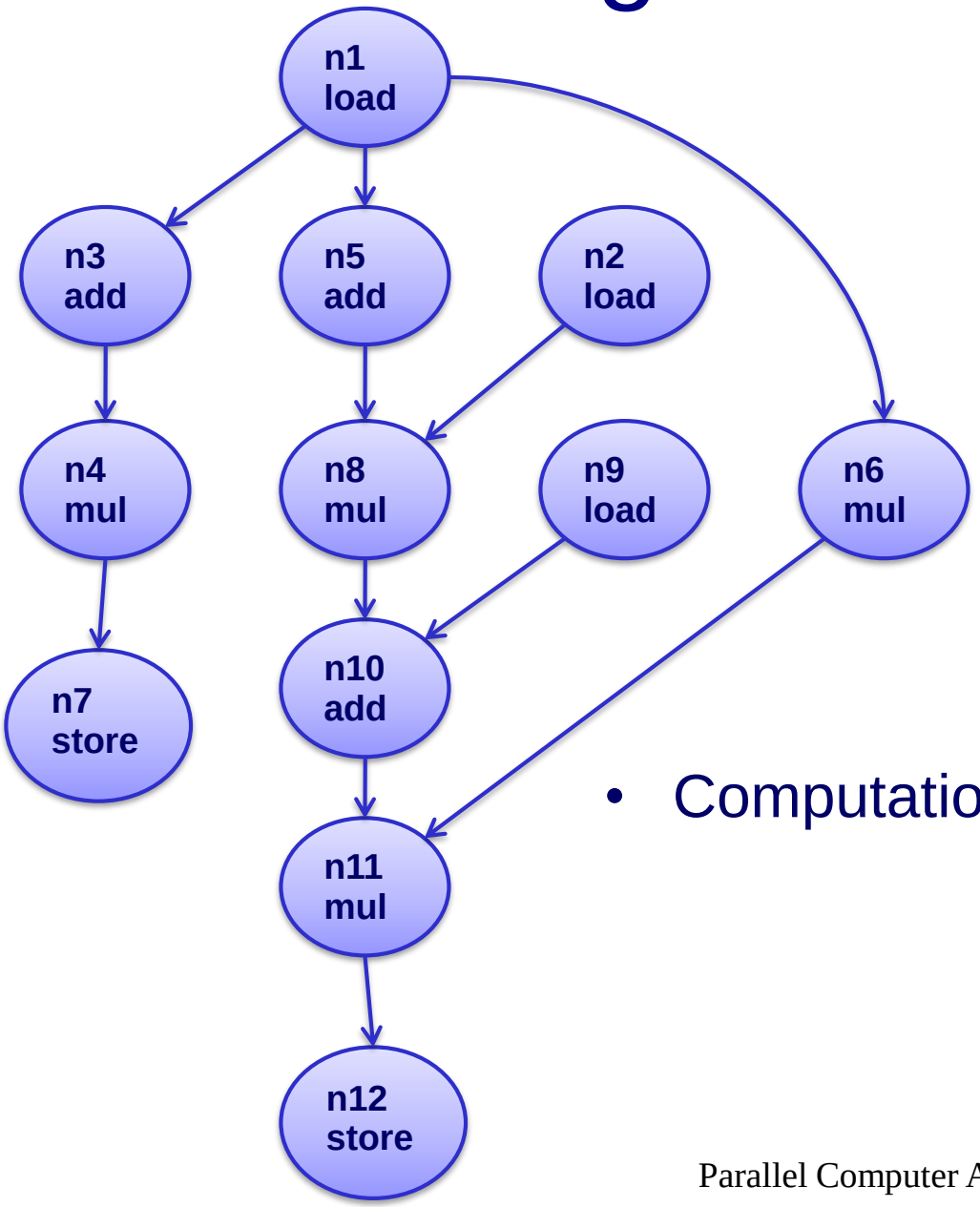
Recurrence Constraints

- Lower bound of the II due to all loops is

$$\mathrm{Re}\,cII = \max_{loops} \left| M_i / D_i \right|$$

- Finally: $MII = \max(\mathrm{Re}\,sMII, \mathrm{Re}\,cMII)$

Swing Modulo Scheduling



- Processor model
  - 2 load/store units
  - 1 add unit
  - 1 mul unit
- Latencies
  - add: 2 cycles
  - mul: 2 cycles
  - load: 2 cycles
  - store: 1 cycle

- Computation of minimum iteration interval (II)

$$\mathrm{ResMII} = \max\left(\left\lceil \frac{5}{2} \right\rceil, 3, 4\right) = 4$$

$$\mathrm{RecMII} = 0$$

$$\mathrm{MII} = \max(\mathrm{ResMII}, \mathrm{RecMII}) = 4$$

Swing Modulo Scheduling (Definitions I)

- First, the definition of some terms based on the CDFG:
  - $ASAP_u$ (As Soon As Possible) : the earliest time at which node u can be scheduled:

$$ASAP_u = \begin{cases} 0, \text{if } \text{Pred}_u = NULL \\ \max(ASAP_v + \lambda_v), \forall v \in \text{Pred}_u \end{cases}$$

  - $ALAP_u$ (As Late As Possible) : the latest time at which node u can be scheduled:

$$ALAP_u = \begin{cases} \max(ASAP_v), \forall v \in V, \text{if } \text{Succ}_u = NULL \\ \min(ALAP_v - \lambda_u), \forall v \in \text{Succ}_u \end{cases}$$

  - MOV_u (Mobility Function) : the number of slots in which node u can be legally scheduled:

$$MOV_u = ALAP_u - ASAP_u$$

Swing Modulo Scheduling
(Definitions II)

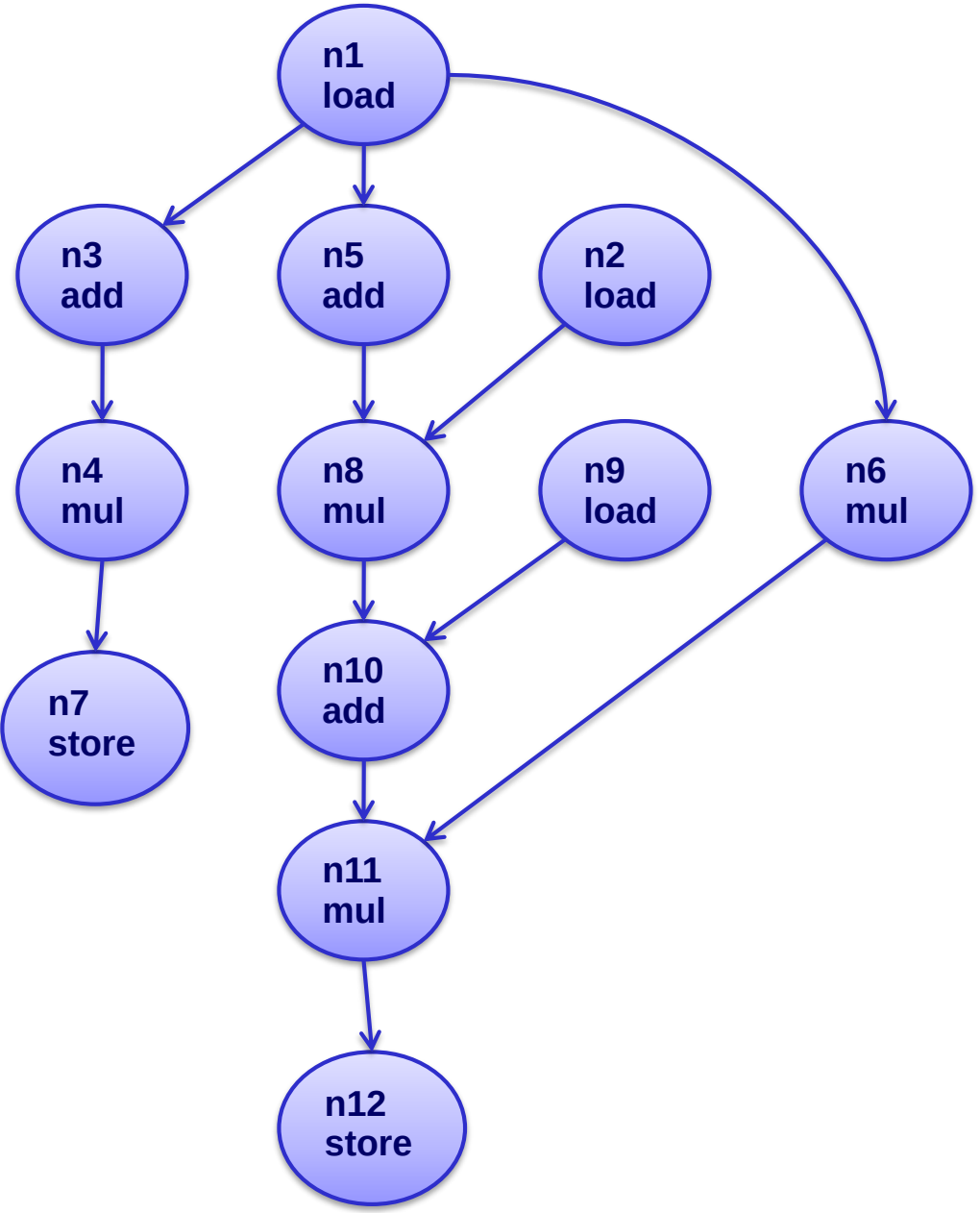
- D_u (Depth) : the maximum number of predecessors of node u weighted by their latency :

$$D_u = \begin{Bmatrix} 0, \text{if } \text{Pred}_u = NULL \\ \max(D_v + \lambda_v), \forall v \in \text{Pred}_u \end{Bmatrix}$$

- H_u (Height) : the maximum number of successors of node u weighted by their latency :

$$H_u = \begin{Bmatrix} 0, \text{if } \text{Succ}_u = NULL \\ \max(H_v + \lambda_v), \forall v \in \text{Succ}_u \end{Bmatrix}$$

Swing Modulo Scheduling: Example



node	ASAP	ALAP	MOV	D	H
n1	0	0	0	0	10
n2	0	2	2	0	8
n3	2	6	4	2	4
n4	4	8	4	4	2
n5	2	2	0	2	8
n6	2	6	4	2	4
n7	6	10	4	6	0
n8	4	4	0	4	6
n9	0	4	4	0	6
n10	6	6	0	6	4
n11	8	8	0	8	2
n12	10	10	0	10	0

Node Ordering Algorithm

- Basic ideas

1. Give priority to operations in the most critical paths. Schedule those first.
   - These are the paths whose nodes have small MOV
2. Try to minimize the number of live variables in every cycle of the schedule
   - To reduce the required number of registers to store node outputs

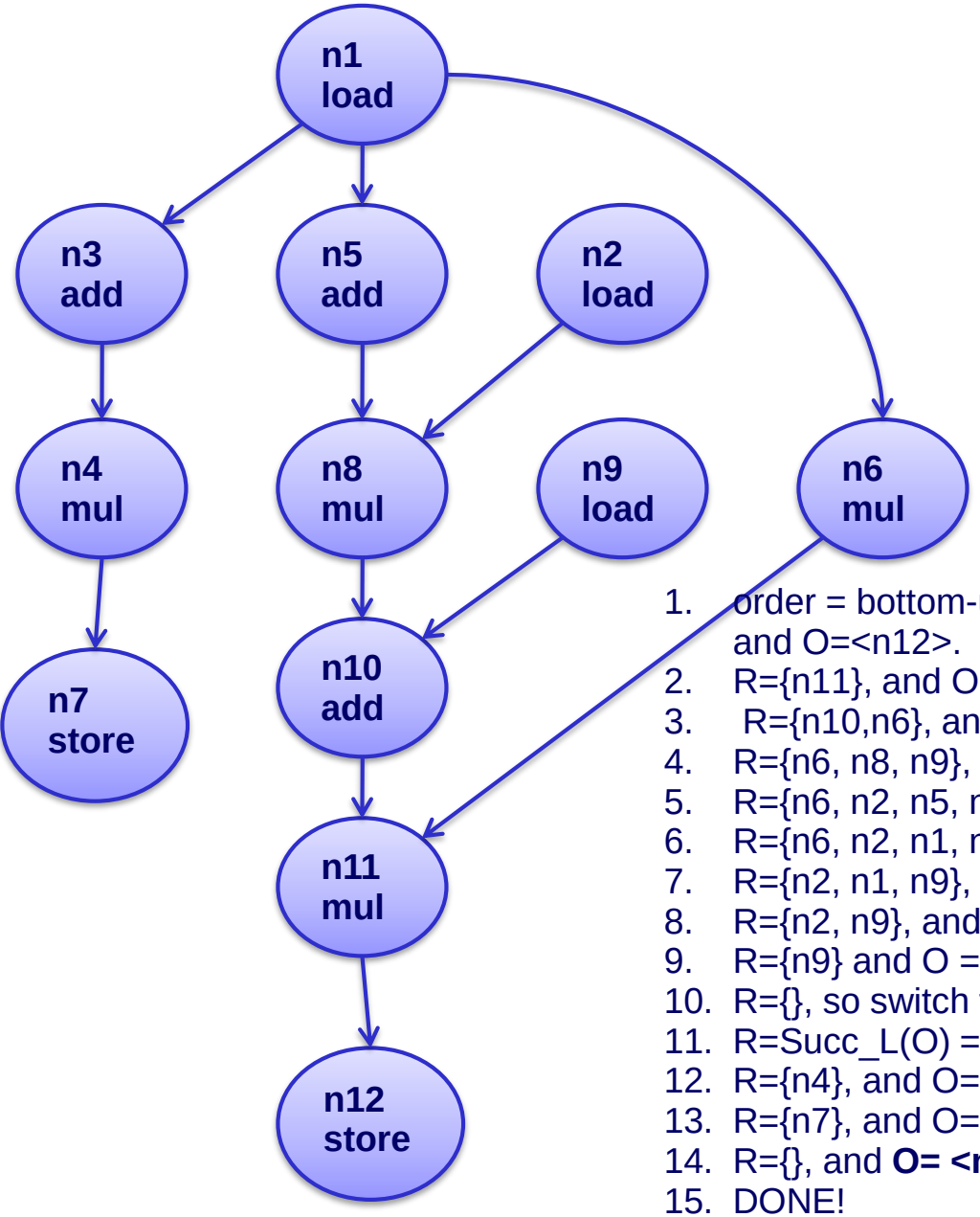
Node Ordering Algorithm

Start from the node at the bottom of the CDFG.
Move bottom-up visiting all ancestors.
Once all ancestors have been visited, start moving top-down
Visit all descendants in the order given by their height.

We keep on moving up and down.

```
Repeat
   if order = top-down
      while R ≠ ∅ do
         v := Element of R with the highest H_v ;
               if more than one, choose node with lowest MOV_L
         O := O | <v>
         R := R - {v} ∪ (Suc (v) ∩ S)
      endwhile
      order := bottom-up
      R := Pred_L(O) ∩ S
   else
      while R ≠ ∅ do
         v := Element of R with the highest D_v ;
               if more than one, choose node with lowest MOV_L
         O := O | <v>
         R := R - {v} ∪ (Pred(v) ∩ S)
      endwhile
      order := top-down
      R := Suc_L(O) ∩ S
   endif
until R = ∅
endfor
```

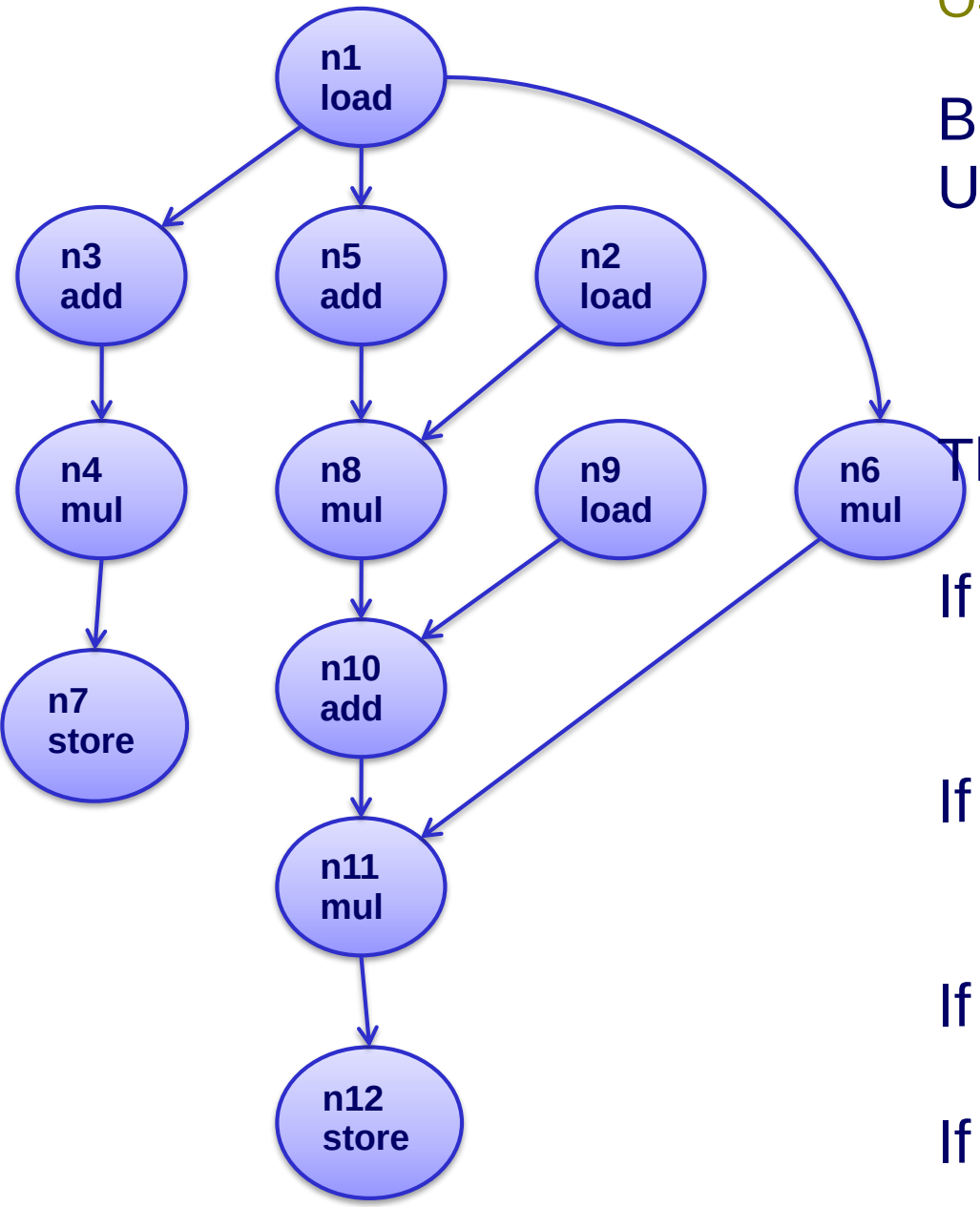
Node Ordering Algorithm



node	ASAP	ALAP	MOV	D	H
n1	0	0	0	0	10
n2	0	2	2	0	8
n3	2	6	4	2	4
n4	4	8	4	4	2
n5	2	2	0	2	8
n6	2	6	4	2	4
n7	6	10	4	6	0
n8	4	4	0	4	6
n9	0	4	4	0	6
n10	6	6	0	6	4
n11	8	8	0	8	2
n12	10	10	0	10	0

1. order = bottom-up, R={n12}, because it has the largest depth D(n12)=10, and O=<n12>.
2. R={n11}, and O = <n12, n11>
3. R={n10,n6}, and O = <n12,n11,n10>
4. R={n6, n8, n9}, and O = <n12,n11,n10,n8>
5. R={n6, n2, n5, n9}, and O = <n12,n11,n10,n8,n5>
6. R={n6, n2, n1, n9}, and O = <n12,n11,n10,n8,n5,n6>
7. R={n2, n1, n9}, and O = <n12,n11,n10,n8,n5,n6,n1>
8. R={n2, n9}, and O = <n12,n11,n10,n8,n5,n6,n1,n2>
9. R={n9} and O = <n12,n11,n10,n8,n5,n6,n1,n2,n9>
10. R={}, so switch to top-down
11. R=Succ_L(O) = {n3}, and O = <n12,n11,n10,n8,n5,n6,n1,n2,n9,n3>
12. R={n4}, and O= <n12,n11,n10,n8,n5,n6,n1,n2,n9,n3,n4>
13. R={n7}, and O= <n12,n11,n10,n8,n5,n6,n1,n2,n9,n3,n4,n7>
14. R={}, and **O= <n12,n11,n10,n8,n5,n6,n1,n2,n9,n3,n4,n7>**
15. DONE!

Node Scheduling Algorithm



O= <n12,n11,n10,n8,n5,n6,n1,n2,n9,n3,n4,n7>

Basic idea:

Using the ordering *O*, schedule the nodes as close as possible to neighbors that have already been scheduled.

This way, we reduce register pressure

If a node *u* has only predecessors in the partial schedule, schedule *u* as soon as possible.

If a node *u* has only successors in the partial schedule, schedule *u* as late as possible.

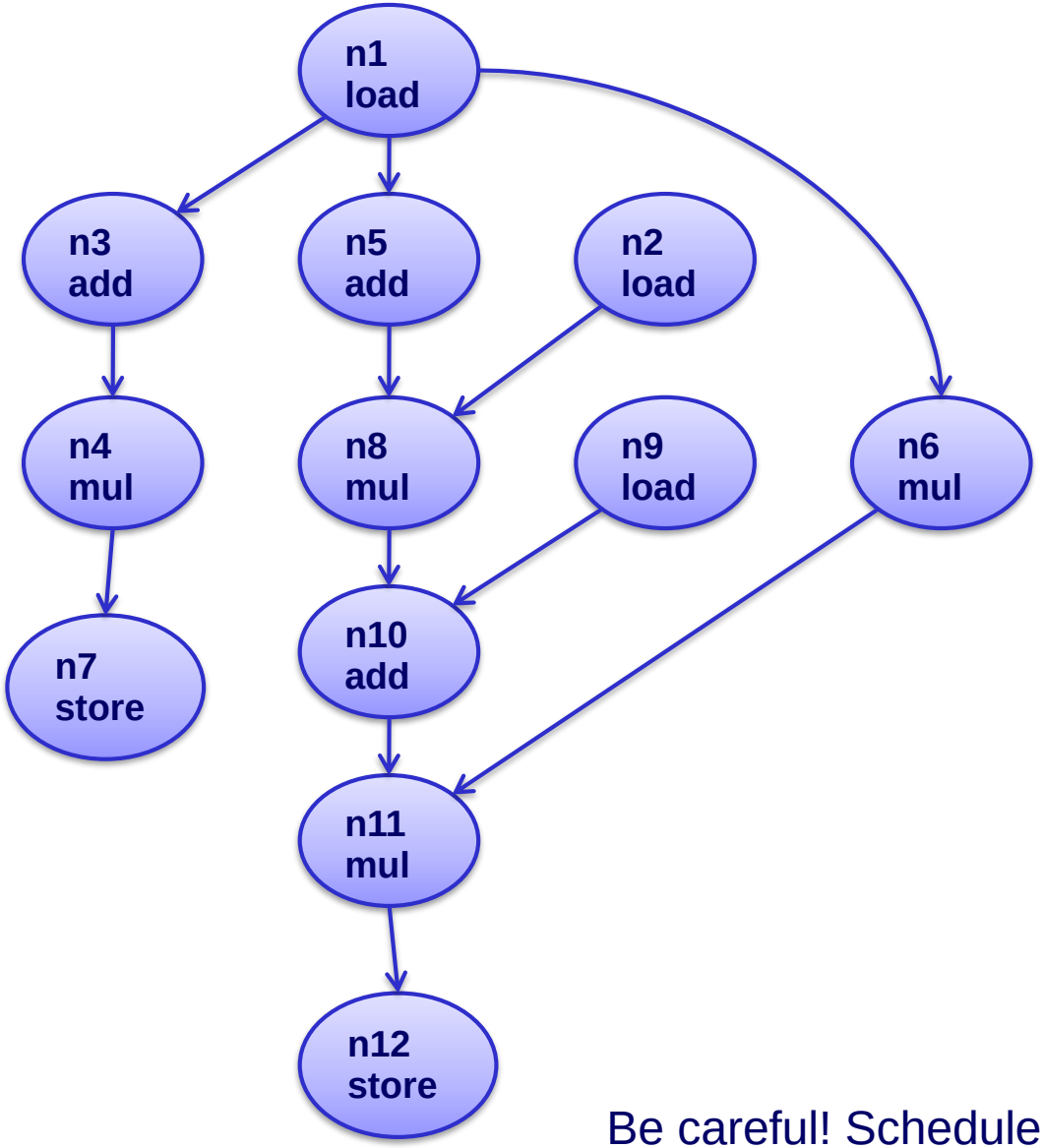
If a node *u* has both, schedule *u* within the range [$ASAP_u$, $ALAP_u$]

If a node *u* has none, schedule *u* in the $ASAP_u$

Node Scheduling Algorithm

II = 4

O= <n12,n11,n10,n8,n5,n6,n1,n2,n9,n3,n4,n7>



	L/S 1	L/S 2	ADD	MUL
0	n1			
1				
2	n2		n5	
3				
4			n3	n8
5		n9		
6				n4
7			n10	n6
8		n7		
9				n11
10				
11	n12			

Be careful! Schedule is modulo II

Node Scheduling Algorithm

	L/S 1	L/S 2	ADD	MUL
0	n1[0]			
1				
2	n2[0]		n5[0]	
3				
4			n3[0]	n8[0]
5		n9[0]		
6				n4[0]
7			n10[0]	n6[0]
8		n7[0]		
9				n11[0]
10				
11	n12[0]			

Final Modulo Schedule

	L/S 1	L/S 2	ADD	MUL
0	n1[i+2]	n7[i]	n3[i+1]	n8[i+1]
1		n9[i+1]		n11[i]
2	n2[i+2]		n5[i+2]	n4[i+1]
3	n12[i]		n10[i+1]	n6[i+1]

Implementation issues

- One of the main advantages of modulo scheduling is that the compiler only produces the schedule for the kernel
  - This results into limited code size
- Some implementation complications arise due to this schedule
- Some of these issues are resolved by hardware support

Issue #1 - Overlapped Lifetimes

	L/S 1	L/S 2	ADD	MUL
0	n1			
1				
2	n2		n5	
3				
4	n1		n3	n8
5		n9		
6	n2		n5	n4
7			n10	n6
8	n1	n7	n3	n8
9		n9		n11
10	n2		n5	n4
11	n12		n10	n6

Between the *definition* and the last *use* of the output of node n1 by node n6 there is an additional n1 definition.

Therefore, the scheduled code is incorrect!

Issue #1 - Overlapped Lifetimes

- *Modulo Variable Expansion* is a compiler technique used to solve the problem of overlapping live registers
- Unroll the kernel and rename registers at compile time
- Minimum degree of unroll K_{min}=ceiling(v/II)
- v is length of longest lifetime
- in previous example, k = ceil(7/4) = 2

Issue #1 - Overlapped Lifetimes

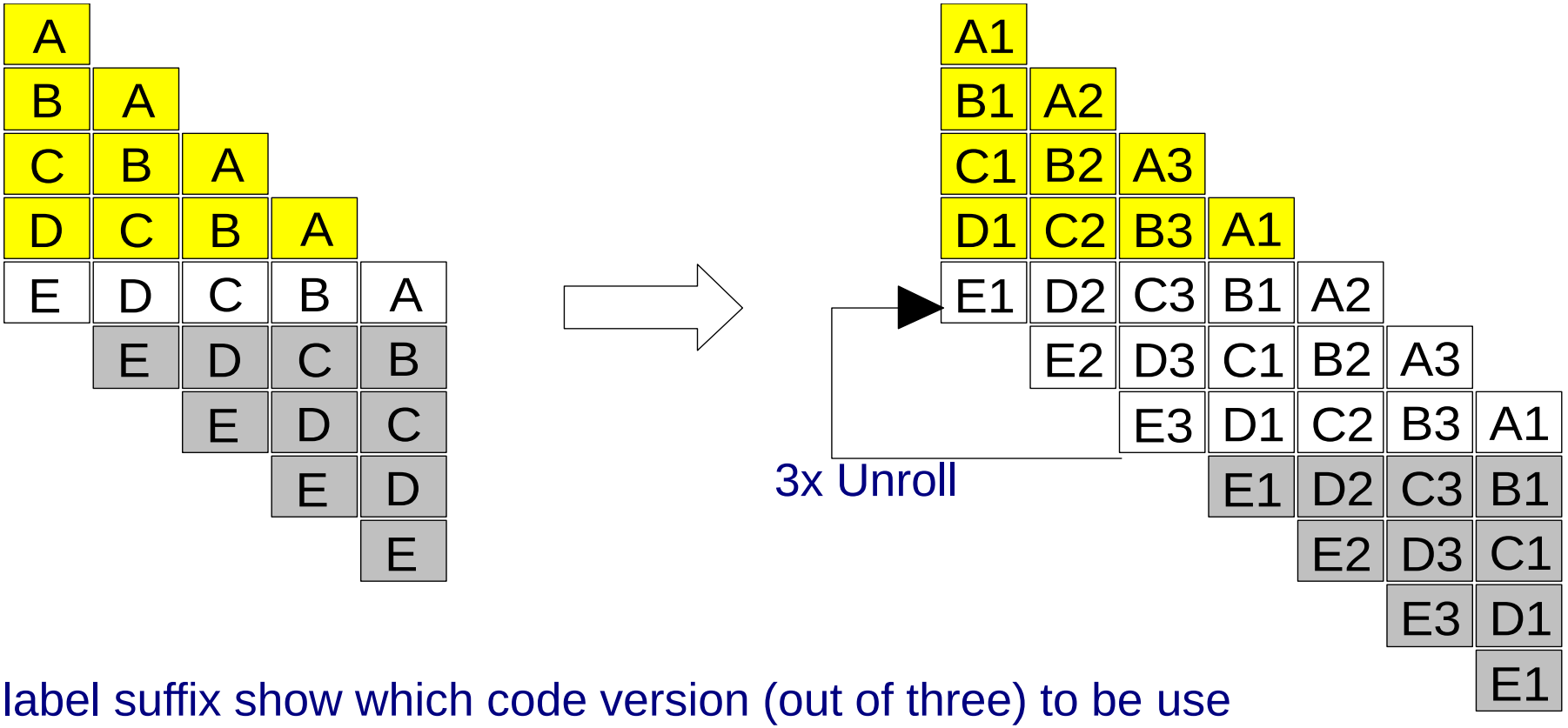
L/S 1	L/S 2	ADD	MUL
n1			
n2		n5	
n1		n3	n8
	n9		
n2		n5	n4
		n10	n6
n1	n7	n3	n8
	n9		n11
n2		n5	n4
n12		n10	n6

	L/S 1	L/S 2	ADD	MUL
0	n1_1			
1				
2	n2_1		n5_1	
3				
4	n1_2		n3_1	n8_1
5		n9_1		
6	n2_2		n5_2	n4_1
7			n10_1	n6_1
8	n1_1	n7_1	n3_2	n8_2
9		n9_2		n11_1
10	n2_1		n5_1	n4_2
11	n12_1		n10_2	n6_2
12	n1_2	n7_2	n3_1	n8_1
13		n9_1		n11_2
14	n2_2		n5_2	n4_1
15	n12_2		n10_1	n6_1

	L/S 1	L/S 2	ADD	MUL
0	n1_1[N]	n7_1[N-2]	n3_2[N-1]	n8_2[N-1]
1		n9_2[N-1]		n11_1[N-2]
2	n2_1[N]		n5_1[N]	n4_2[N-1]
3	n12_1[N-2]		n10_2[N-1]	n6_2[N-1]
4	n1_2[N]	n7_2[N-2]	n3_1[N-1]	n8_1[N-1]
5		n9_1[N-1]		n11_2[N-2]
6	n2_2[N]		n5_2[N]	n4_1[N-1]
7	n12_2[N-2]		n10_1[N-1]	n6_1[N-1]

Steady state schedule with
II=4, Unroll Factor = 2

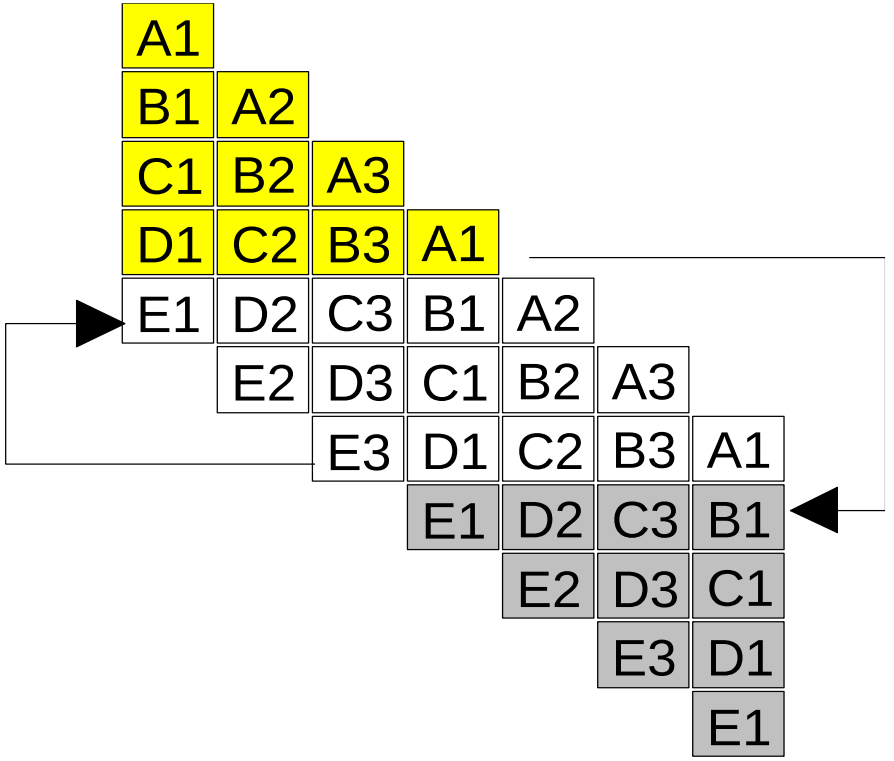
Kernel Unrolling for K=3



The label suffix show which code version (out of three) to be use

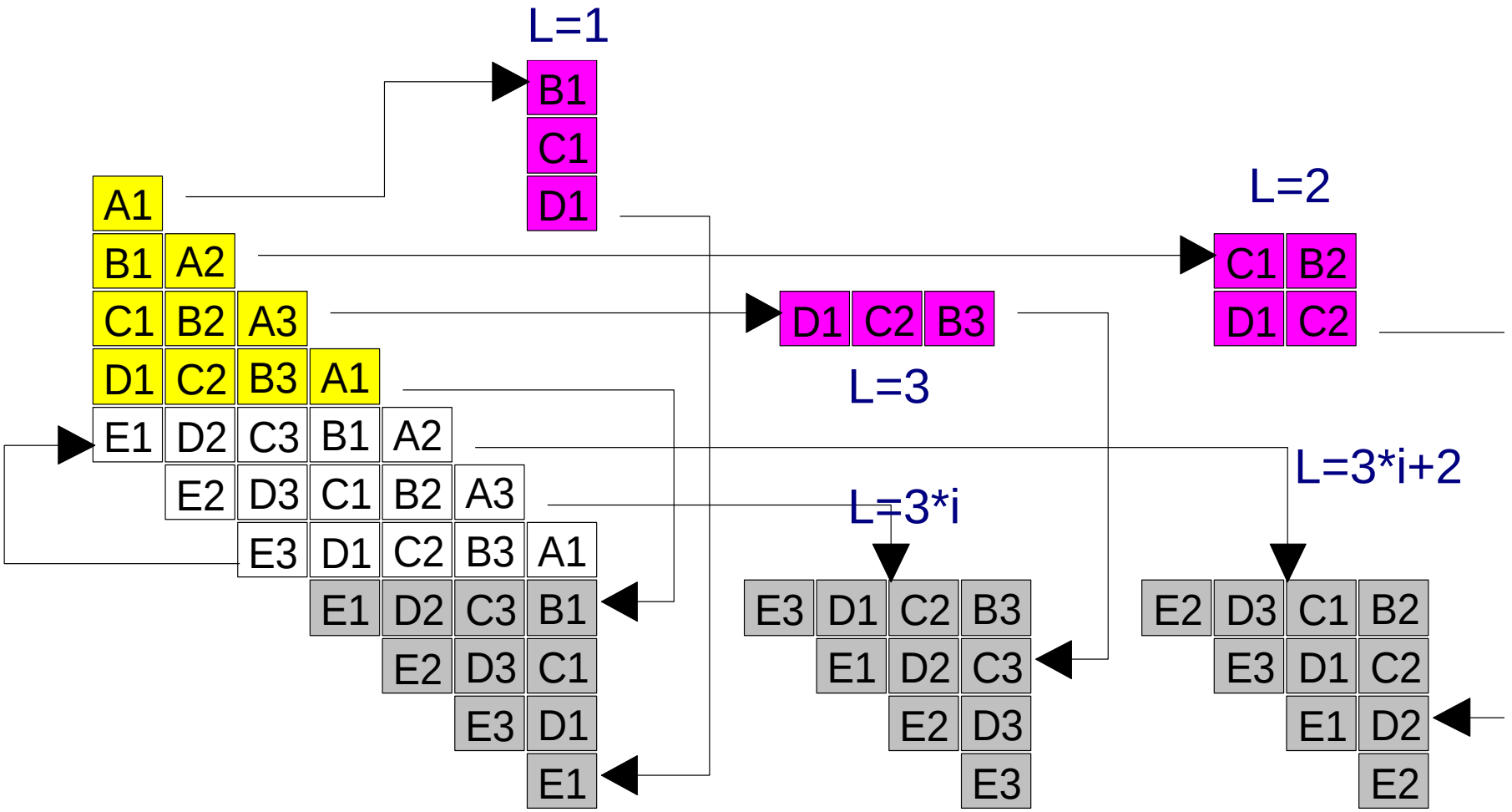
- **Question**: Can we use the register renaming capabilities of Tomasulo's algorithm to resolve overlapping lifetimes?

Issue #2 - Correct Code for All Possible Trip Counts



This scheme works only for 3*i+4 iterations, i>=0
In general, only for K*i + (SC-1) iterations
How do we augment the code for an arbitrary number of iterations L?

Issue #2 - Multiple Epilogues



Issue #3 - Iterative Modulo Scheduling

- The MII may not be an achievable lower bound
  - instructions along a recurrence delayed by resource conflicts
  - complex patterns of resource usage

Iterative Modulo Scheduling

- Increment II till reaching a schedule
- Even if a schedule exists at MII, a list scheduler may fail to find it
  - allow limited unscheduling and rescheduling
  - limit the number of scheduling attempts for each instruction