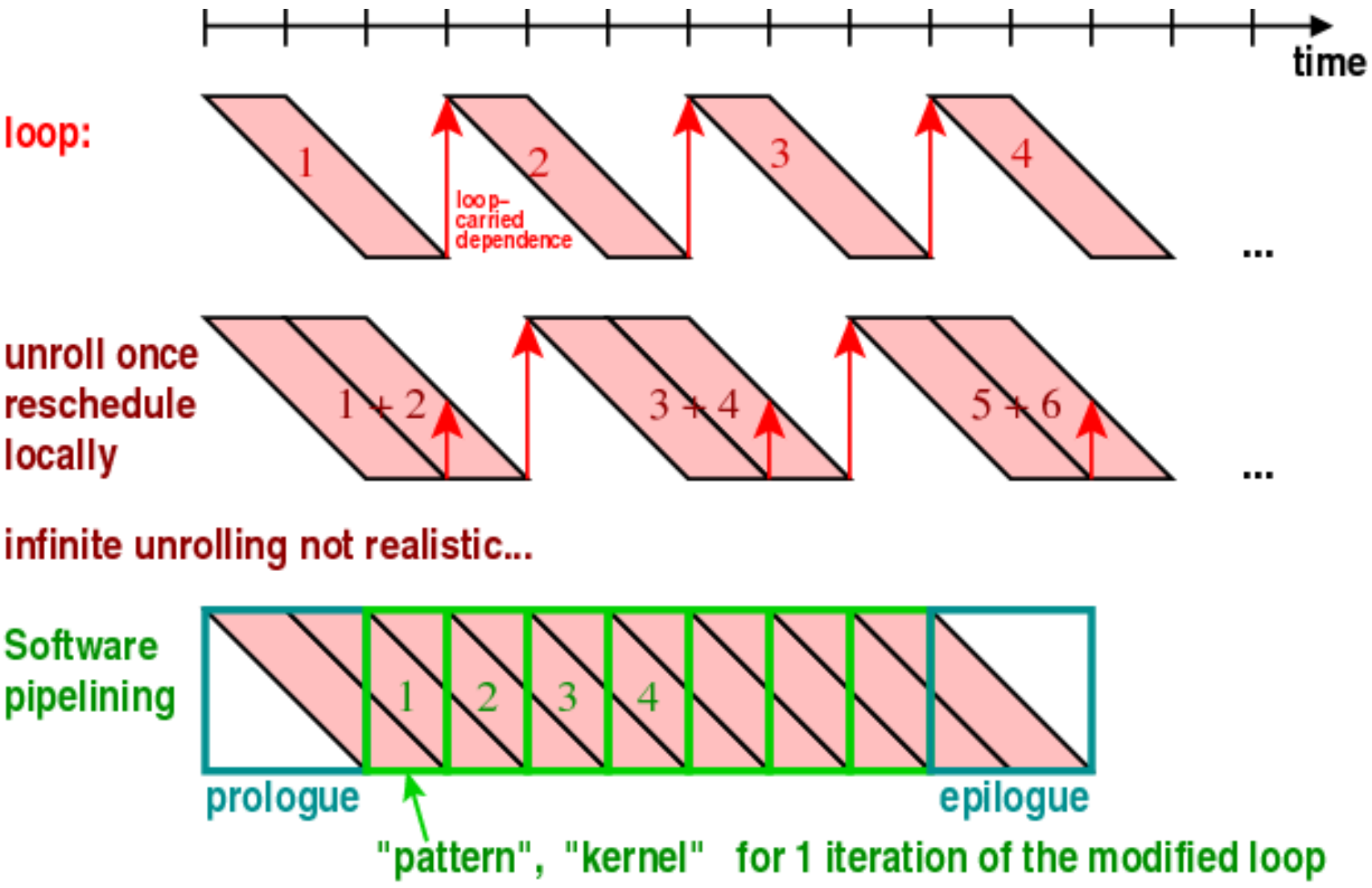
Software Pipelining

Literature:

- C. Kessler, “Compiling for VLIW DSPs”, chapter in *Handbook of Signal Processing Systems*, 3rd edition, 2019 (preprint, handed out)
- ALSU2e Section 10.5
- Muchnick Section 17.4
- V. Allan *et al*.: Software Pipelining. *ACM Computing Surveys* 27(3), 1995

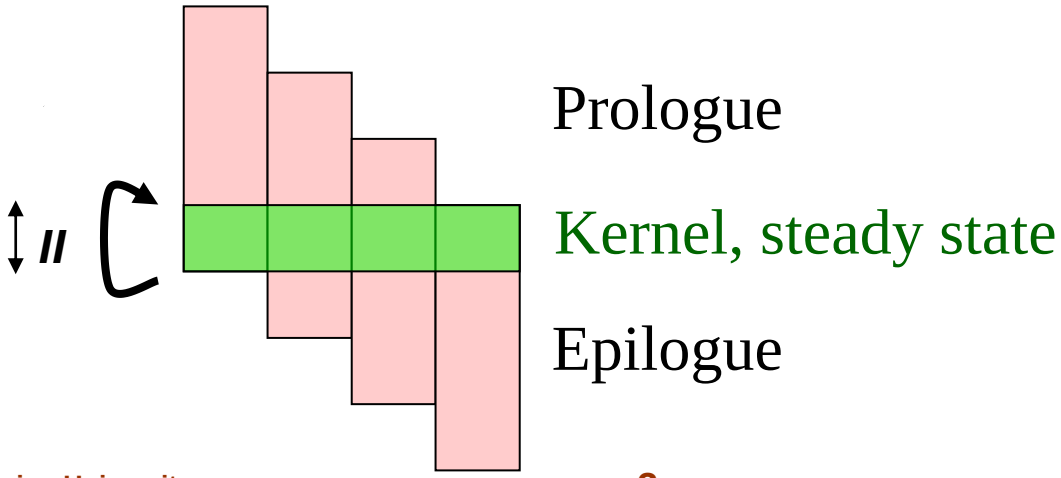
Christoph Kessler, IDA,
Linköping University

Software Pipelining of Loops (1)

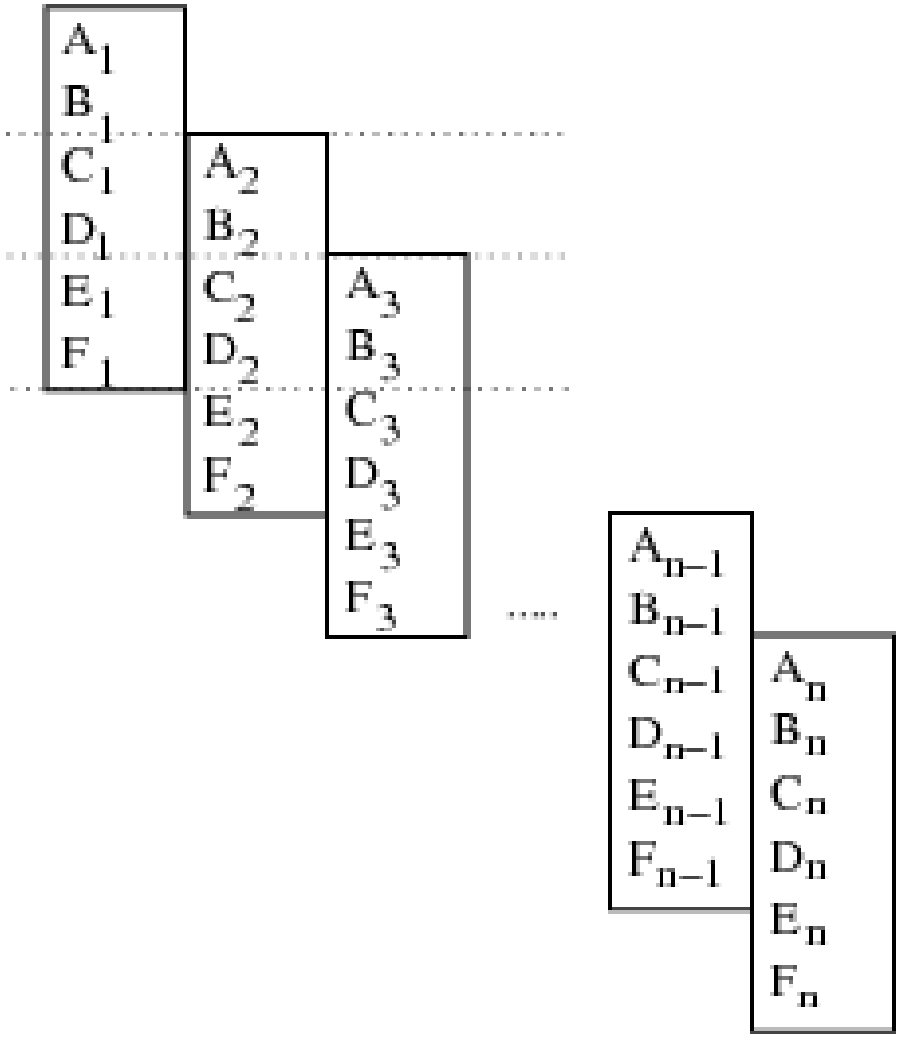


Introduction

- **Software Pipelining** (Modulo Scheduling)
  - Overlap instructions in loops from different iterations
    - Kernel length *II* (initiation interval) ~ Throughput
  - Goal: Faster execution of entire loop
    - Better resource utilization,
    - Increase Instruction Level Parallelism, also in the presence of loop-carried dependences



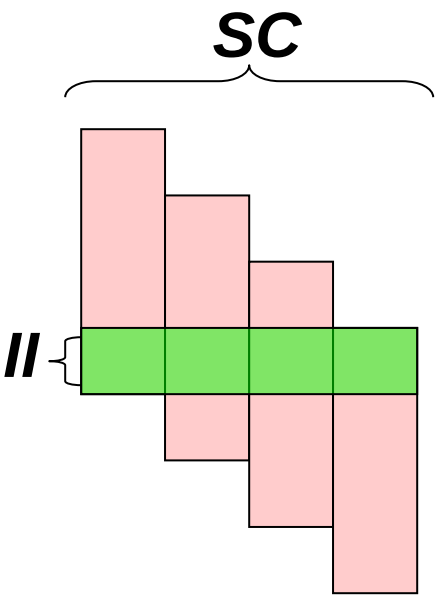
Software Pipelining of Loops (2)



	Unit1	Unit2	Unit3
Prologue:	A_1		
	B_1		
	C_1	A_2	
	D_1	B_2	
	DO i=1, n–2		
Pattern:	E_i	C_{i+1}	A_{i+2}
	F_i	D_{i+1}	B_{i+2}
Epilogue:		E_{n-1}	C_n
		F_{n-1}	D_n
			E_n
			F_n

Definitions

- **Stage count** (*SC*) = makespan for 1 iteration as multiple of kernel lengths
  - degree of overlap / parallelism
  - software pipeline fill/drain overhead (pro-/epilogue)
- **Initiation Interval** (*II*)
- **Minimum Initiation Interval** (MII)
  - Depends on
    - Data dependence cycles (loop carried), ***RecMII***
    - Resources (registers, functional units), ***ResMII***
  - ***MII*** = max(*ResMII, RecMII*)
  - *ResMII* = $\max_U$ ceil(N_U / P),
    - N_U – Number of instructions for resource (functional unit) *U* in the body
    - *P* – Number of functional units



Lower Bound for MII

lower bound on MII: max($ResMII$, $RecMII$)

Resource constraints: $ResMII = \left\lceil \frac{N}{p} \right\rceil$

ignoring data dependences

(multiple reservations: maximize over ratios for each resource type)

Recurrence constraints: (ignoring ressource constraints)

due to (simple) cyclic chains of data dependences (DDG):

accumulate their distances d and their delays l

$$RecMII = \max_{C\ cycle} \left\{ \frac{\sum_{e \in C} l_e}{\sum_{e \in C} d_e} \right\}$$ = maximum overall slope

Calculating the Lower Bound for MII

- max (*ResMII*, *RecMII*)

Determining $ResMII$ is relatively simple.

Determining $RecMII$:

- exhaustive enumeration of all simple cycles
- all-pairs shortest path algorithm (e.g. Floyd-Warshall)
- iterative shortest path algorithm [Zaky'89]
  composing distance matrices for a path algebra $\rightarrow$ trans. closure
- linear programming

More about this in the survey paper by Allan et al.'95

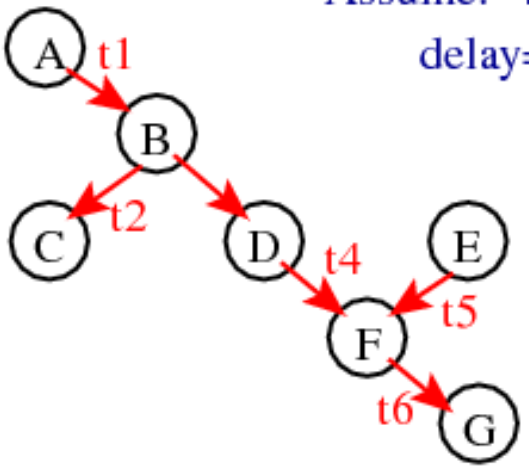
Modulo Scheduling

- **Modulo scheduling**:
  Filling the Modulo Reservation Table,
  one instruction by another

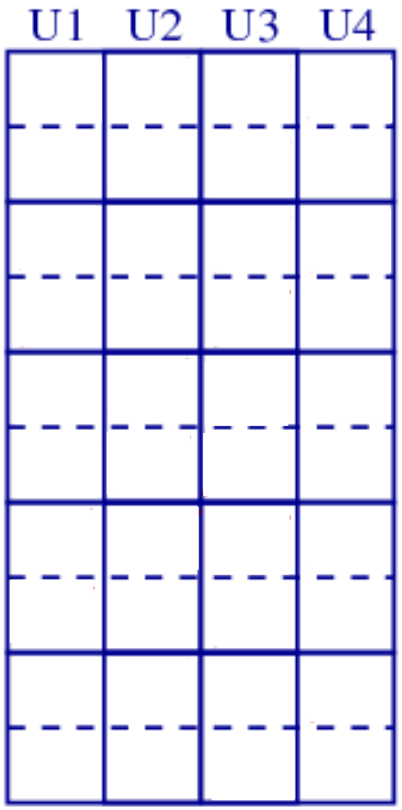
- **Heuristics** → example
  - ASAP, As Soon As Possible
  - ALAP, As Late As Possible
  - HRMS
  - Swing Modulo Scheduling

- **Optimally**
  - Integer Linear Programming [Eriksson'09]

Modulo Scheduling



Assume: 4 units, fully pipelined
delay=2 for all instructions



No dependence cycles

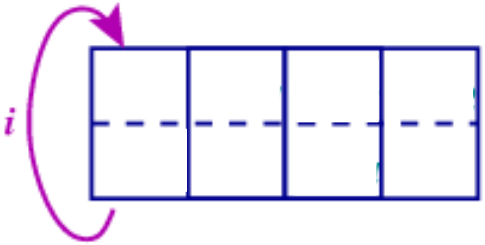
ResMII = ceil(7/4) = 2

Begin with II = ResMII = 2

Apply some local scheduling heuristic
e.g.: list scheduling (A B C D E F G)

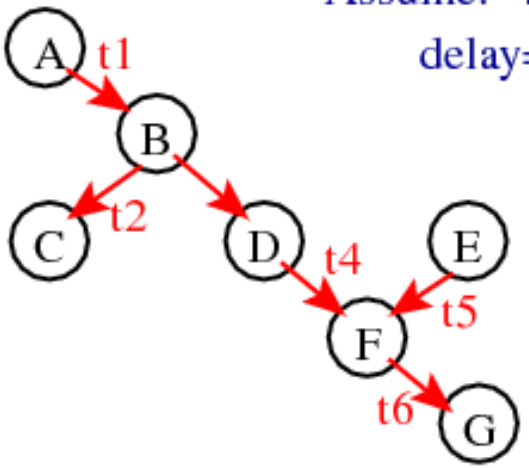
Apply some placement heuristic
e.g.: as early as possible

Mark occupied slots in all iterations...

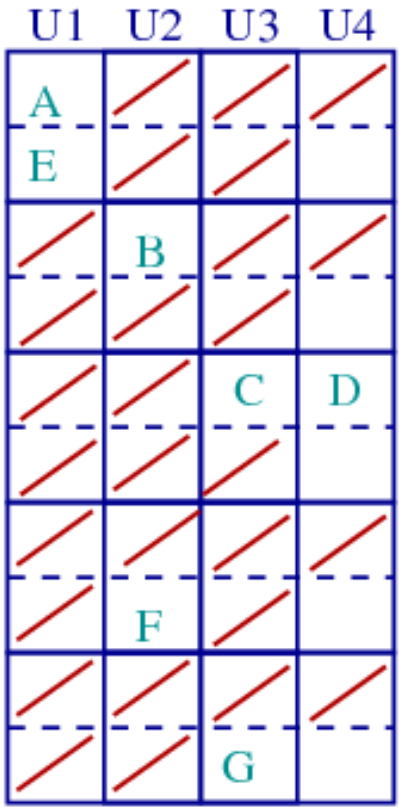


If not possible, increase II and try again...

Modulo Scheduling



Assume: 4 units, fully pipelined
delay=2 for all instructions



No dependence cycles

ResMII = ceil(7/4) = 2

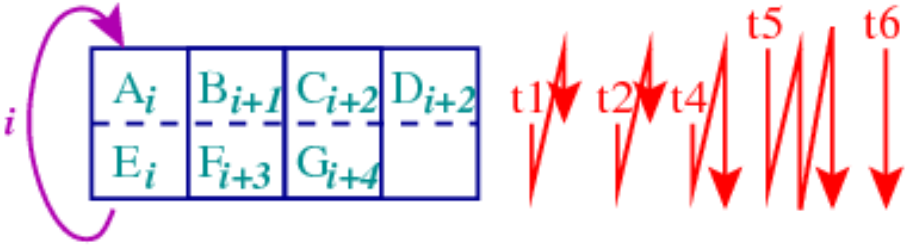
Begin with II = ResMII = 2

Apply some local scheduling heuristic
e.g.: list scheduling (A B C D E F G)

Apply some placement heuristic
e.g.: as early as possible

Mark occupied slots in all iterations...

If not possible, increase II and try again...



Modulo Scheduling Heuristics

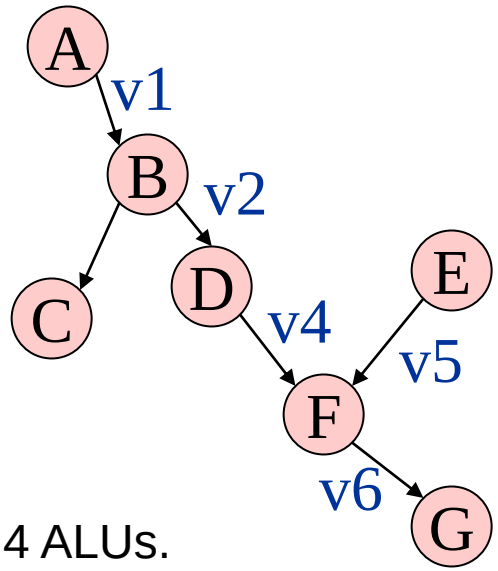
- Example:
  Hypernode Reduction Modulo Scheduling (HRMS)

HRMS – Motivation

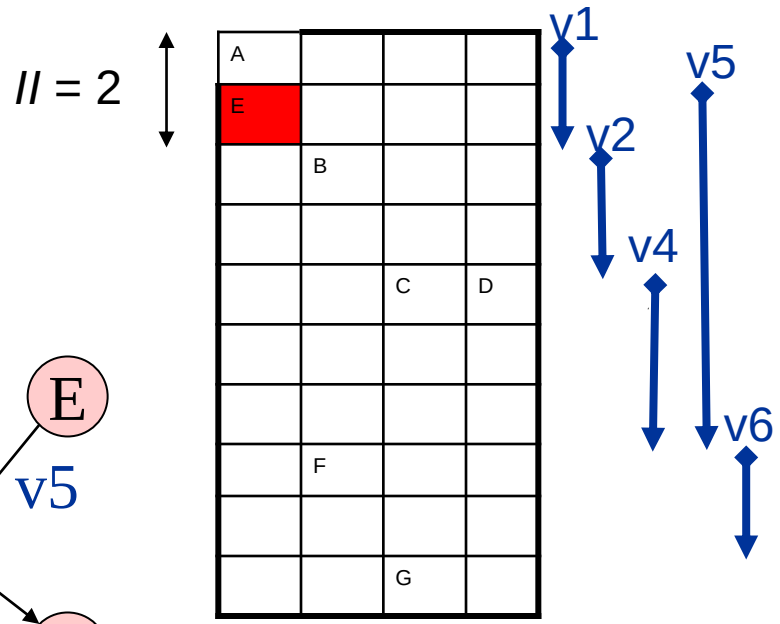
- Problem with simple ASAP or ALAP heuristics:

  Some nodes in the DAG are scheduled too early and some too late

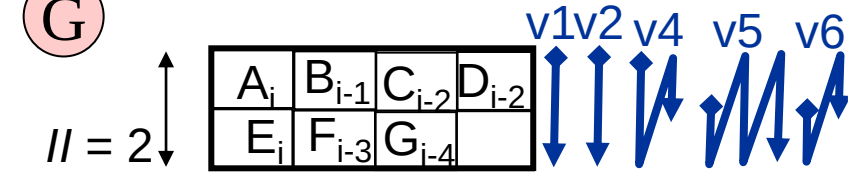
- Example:



Forward order / ASAP



4 ALUs.
Latency = 2
for all operations



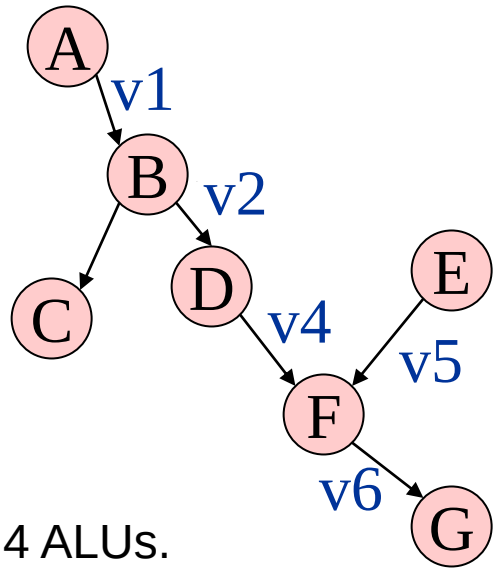
MaxLive = 7

HRMS – Motivation

- Problem with simple ASAP or ALAP heuristics:

  Some nodes in the DAG are scheduled too early and some too late

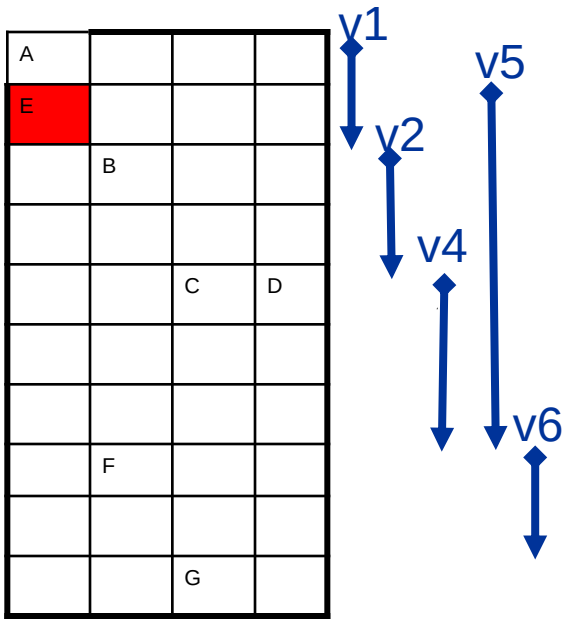
- Example:



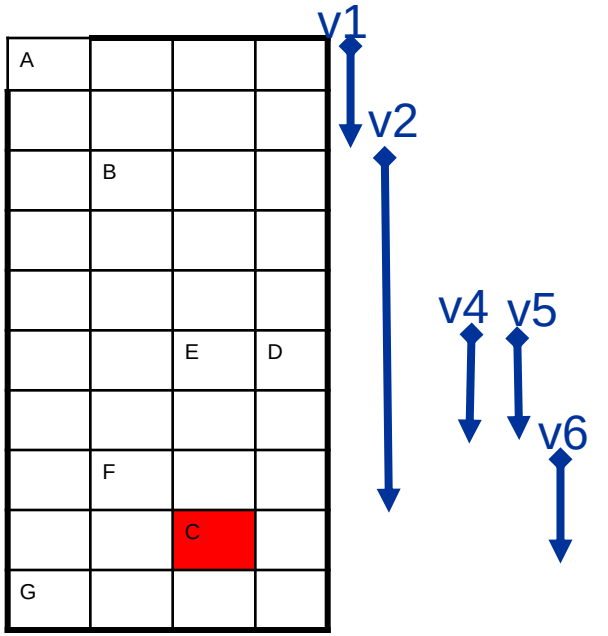
4 ALUs.
Latency = 2
for all operations

II = 2

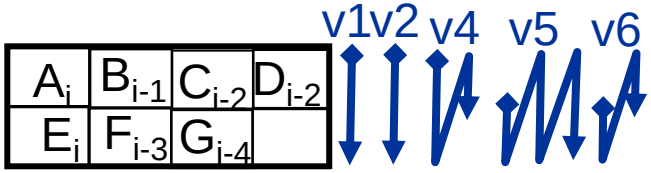
Forward order / ASAP



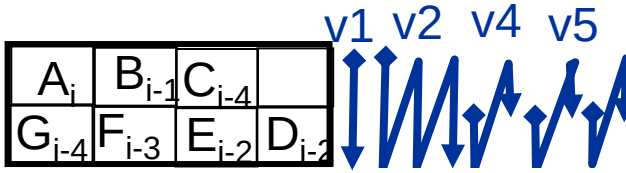
Backward order/ALAP



II = 2



MaxLive = 7



MaxLive = 7

Hypernode Reduction Approach

- Schedule only nodes that have
  - Only predecessors already scheduled *or*
  - Only successors already scheduled *or*
  - None of them,

  but not both predecessors and successors.
- Ensures low register pressure by *scheduling nodes as close as possible to their relatives.*

HRMS – Solution

Two stage algorithm

1. Pre order the nodes of the DAG
   - By using a reduction algorithm
2. Schedule according to the order given in step 1

Pre-Ordering Step

- Select initial node v
  - the *hypernode* $H \leftarrow \{v\}$
- Reduce nodes to the hypernode iteratively
  - Remove iteratively edges and nodes in the DAG (= reducing the DAG) and add them to *H*
- In each reduction step, append to list of ordered set of nodes

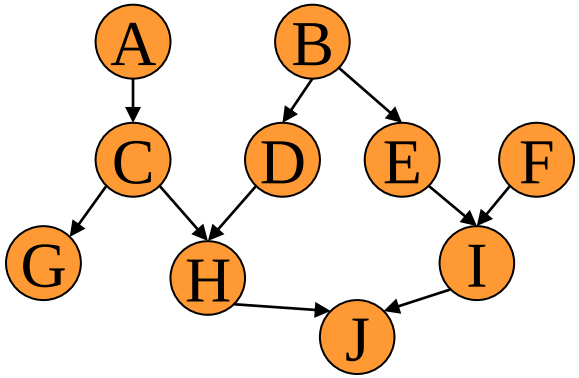
- Similar to list scheduling / topological sorting, but now in both directions – forward and backward along edges incident to *H*

Function pre_ordering(G)

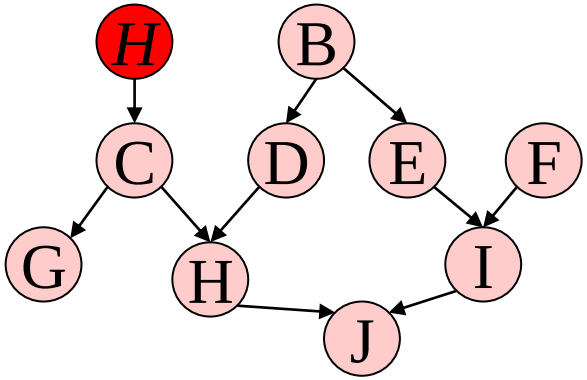
```
Select initial node;    H ← {Initial node};
List = < Initial node >;
While (Pred(H) nonempty or Succ(H) nonempty) do
   V' = Pred(H);
   V' = Search_All_Paths( V',G );
   G' = Hypernode_Reduction( V', G, H );
   L' = Sort_PALA(G');                    // ALAP with inverted order
   List = Concatenate ( List, L' )

   V' = Succ(h);
   V' = Search_AllPaths(V',G);
   G' = Hypernode_Reduction(V',G,h);
   L' = Sort_ASAP(G');
   List = Concatenate ( List, L' );
end while
return List;
```

HRMS – Example

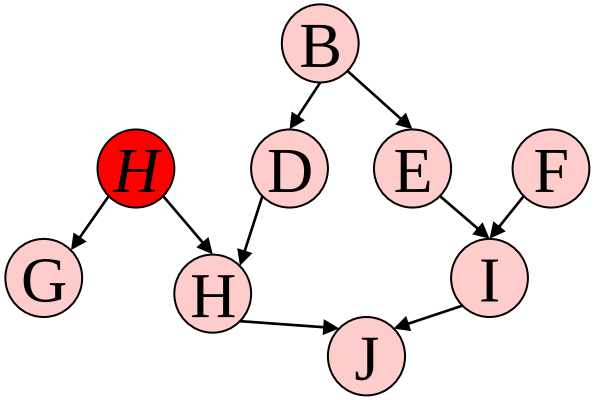
Original dependence graph of one loop iteration:

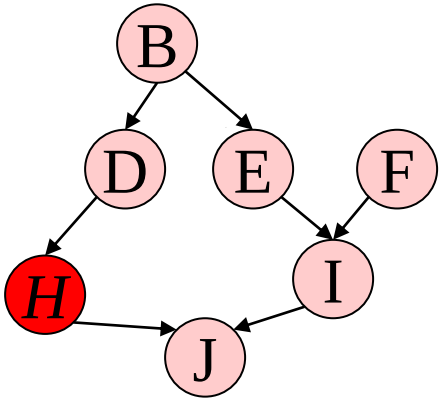


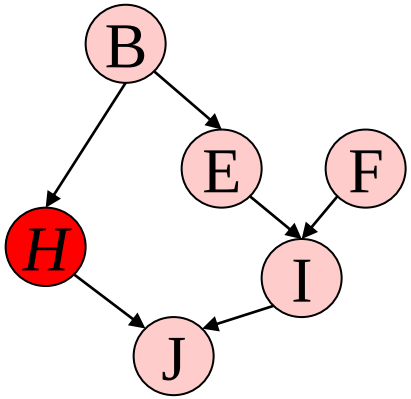
Start with initial hypernode $H = \{ A \}$:

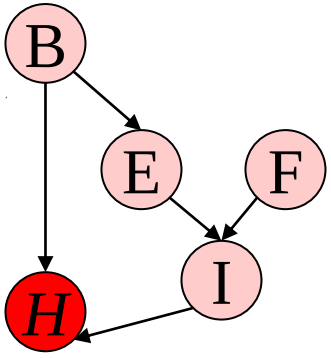


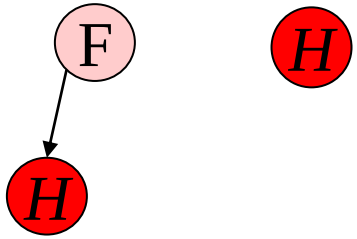
H ”eats” successor node C:











List = { A,C,G,H,D,J,I,E,B,F } Pred nodes to be scheduled ALAP (D,I,E,B,F), Succ nodes scheduled ASAP (A,C,G,H,J)

Pre-Ordering with circular dependencies

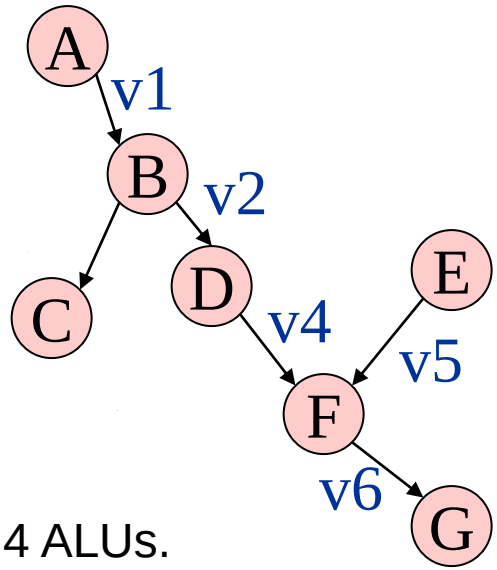
- Circular dependences from loop carried dependences.
- Solution:
  Reduce complete path causing cycle to the Hypernode
- How to deal with several connected cycles in DAG?
  - (See details in the paper, skipped).

The Scheduling Step

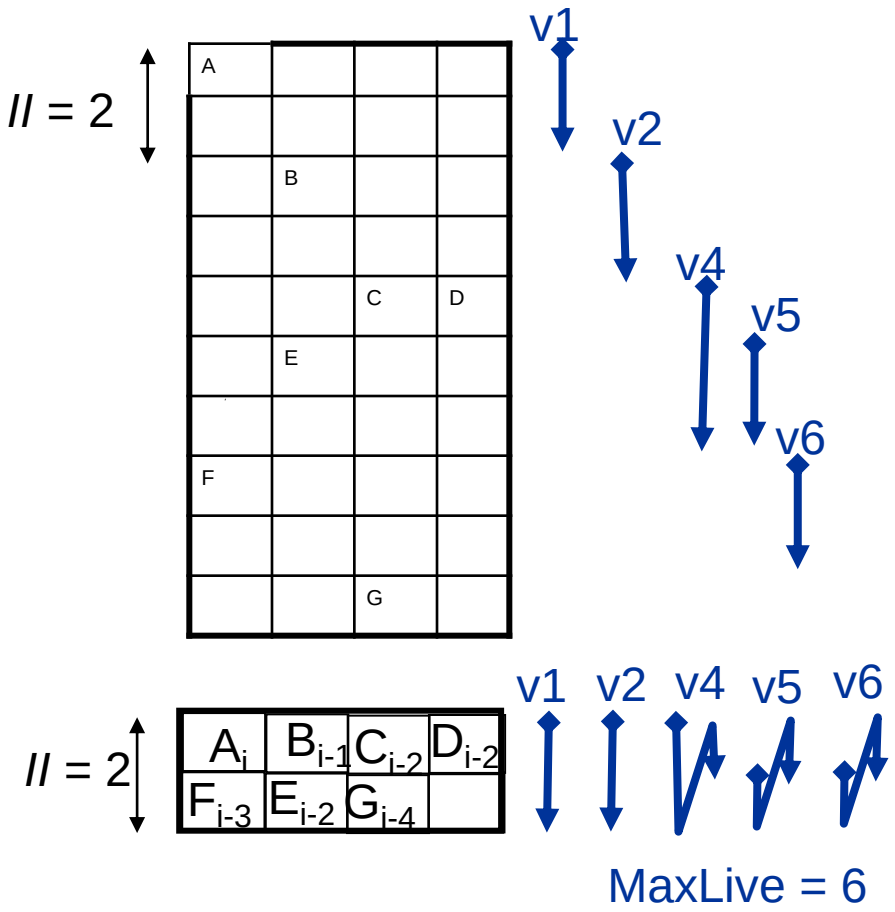
- Places operations in the order given by the pre-ordering step
- Different strategy depending on *neighbors*
- If operation has
  - Only predecessors in partial schedule → **ASAP**
  - Only successors in partial schedule → **ALAP**
  - Both predecessors and successors in partial schedule
    → Scan from ASAP schedule time towards ALAP time.
    - (If no slot found, *II*++ and reschedule)

HRMS – Example (cont.)

- Resulting HRMS schedule:
  - A, B, C, D, E, F, G where for E: ALAP, for others ASAP



4 ALUs.
Latency = 2
for all operations



HRMS – Results

- Perfect Club Benchmark
  - 97.4 % of the loops gave optimal *II*
- Comparison with other algorithms in the paper
  - Works better than Slack scheduling and FRLC scheduling (references in the paper)
  - About same performance as SPILP (optimal algorithm using Integer Linear Programming, ILP) but lower computational complexity

HRMS – Conclusion

- Works well for loops with high register pressure
- Low time complexity
- Tested on large benchmark suite.

Reference:

J. Llosa, M. Valero, E. Ayguadé and A. Gonzáles:
Hypernode Resource Modulo Scheduling.
Proc. 28th ACM/IEEE Int. Symposium on Microarchitecture,
pp. 350-360, IEEE Computer Society Press, 1995

Modulo Scheduling with Recurrences?

- If there are recurrences in the dependence graph: [Lam'88]

  (a) find SCC's

  (b) find cycle in SCC with longest accumulated distance

  (c) schedule each SCC individually
  and collapse cycle, creating a single superinstruction

  (d) apply list scheduling to resulting acyclic graph

- similarly: collapse **if..then..else** statements

- disadvantage: separate schedules of SCCs may not fit well together

Modulo Scheduling and Register Allocation

Live ranges may span over multiple iterations $\rightarrow$ high register need!

If register allocation fails:

(a) try again (with new placement strategy or new scheduling order)

(b) spill some live ranges (add spill code) and restart.

Register need can be optimized:

- modify order in which the instructions are placed
- use other placement strategies,
  e.g. “as late as possible” if already a successor was placed

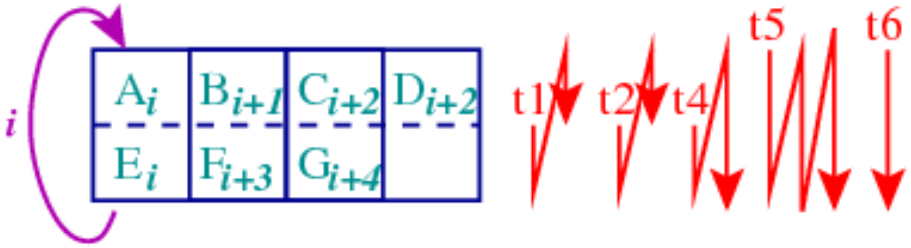
Modulo Scheduling and Register Allocation

- Software Pipelining tends to increase register pressure
  - Live ranges may span over several iterations

- May lead to (more) register spill
  - Introduces new problems
    - Should we spill or increase *II* ?
    - How to choose variables to spill ?
    - Integrated software pipelining (later)

Register Allocation for Modulo-Scheduled Loops

- We call a live range ***self-overlapping*** if it is longer than *II*
  - Needs > 1 physical register
  - Hard to address properly without HW support



- **Modulo Variable Expansion**
  - Unroll the kernel and rename symbolic registers until no self-overlapping live ranges remain
- **A-priori avoidance of self-overlapping live ranges**
  - by live range splitting (inserting copy operations before modulo scheduling) [Stotzer,Leiss LCTES-2009]
- **Hardware support**: **Rotating Register Files**
  - *Iteration Control Pointer* points to window in cyclic loop register file, advanced by hardware loop control

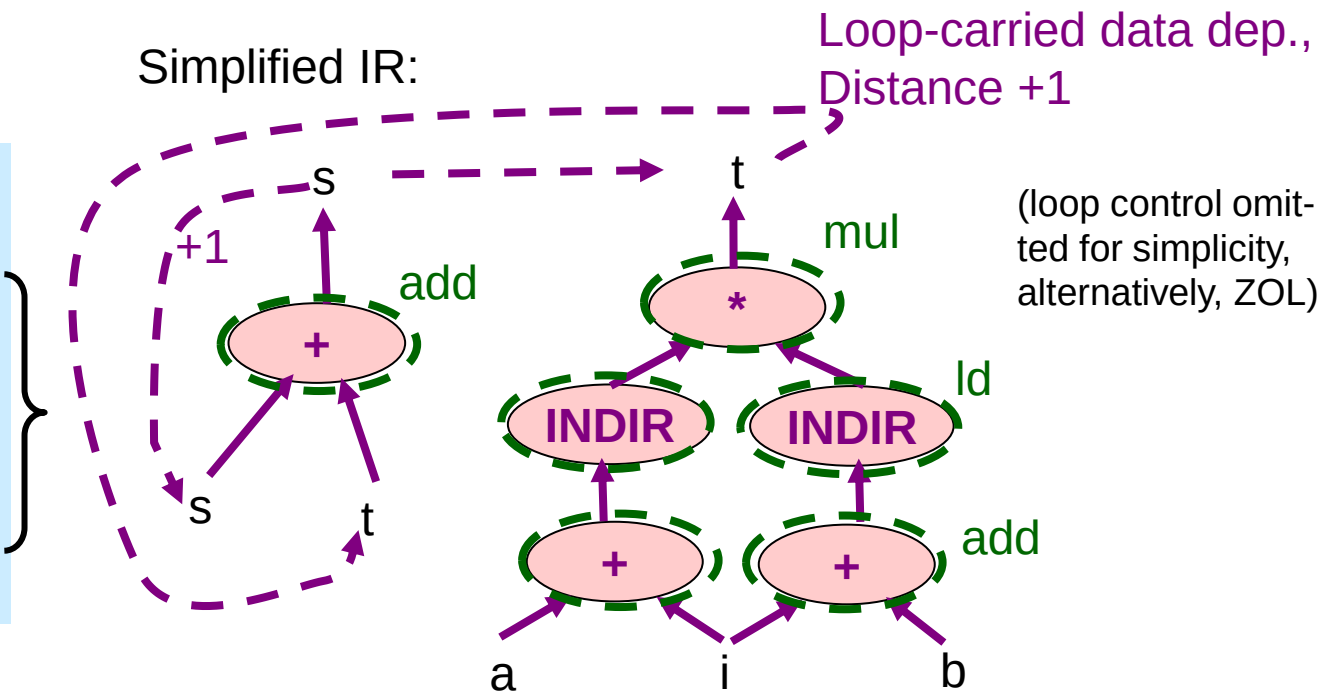
Modulo Scheduling for Loops

at target level

- Example:

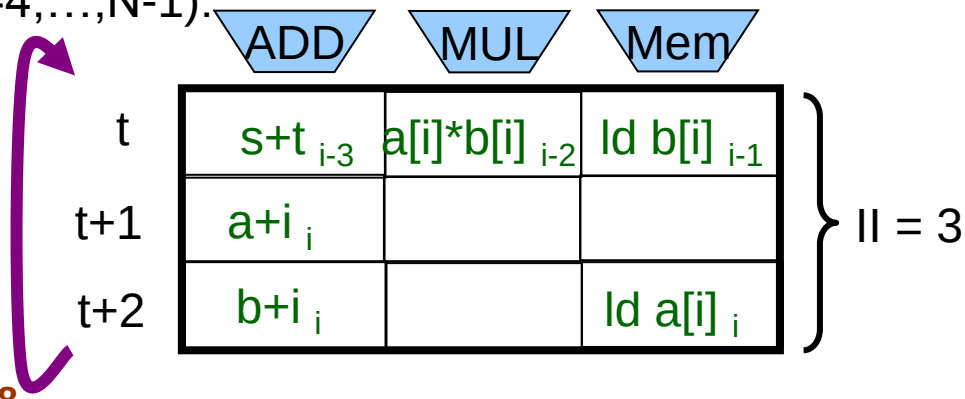
```
s = 0.0;
t = a[0]*b[0];
for ( i=1; i<N; i++)
{
    s = s + t;
    t = a[i] * b[i];
}
s = s + t;
```

Simplified IR:



Given:
VLIW-Processor with 3 units:
- Adder (Latency 1),
- Multiplier/MAC (Latency 3),
- Memory access unit (Latency 3)

Kernel
(i=4,…,N-1):



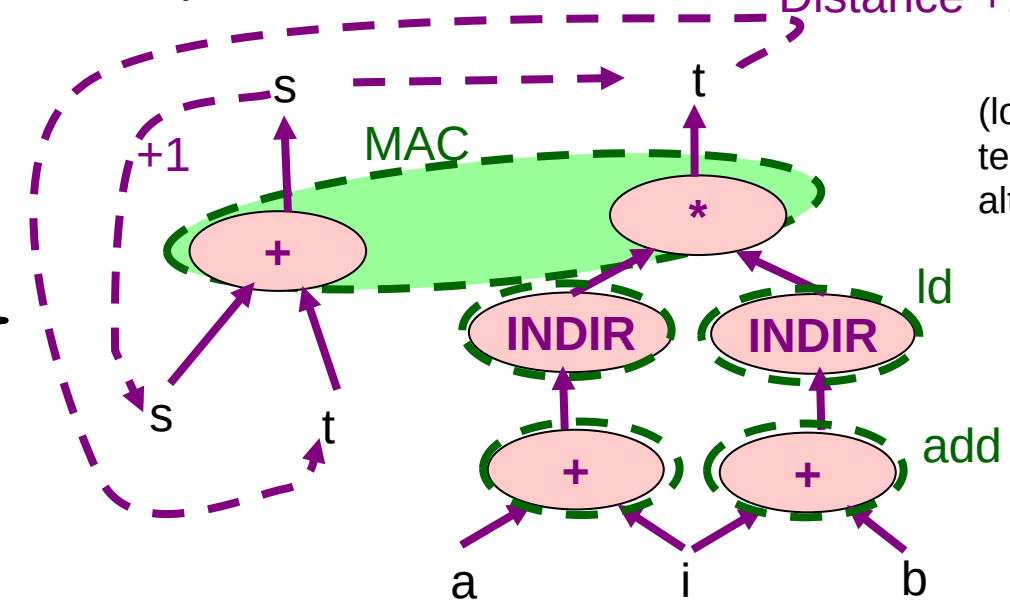
Modulo Scheduling for Loops

can benefit from integration with instruction selection

- Example:

```
s = 0.0;
t = a[0]*b[0];
for ( i=1; i<N; i++)
{
    s = s + t;
    t = a[i] * b[i];
}
s = s + t;
```

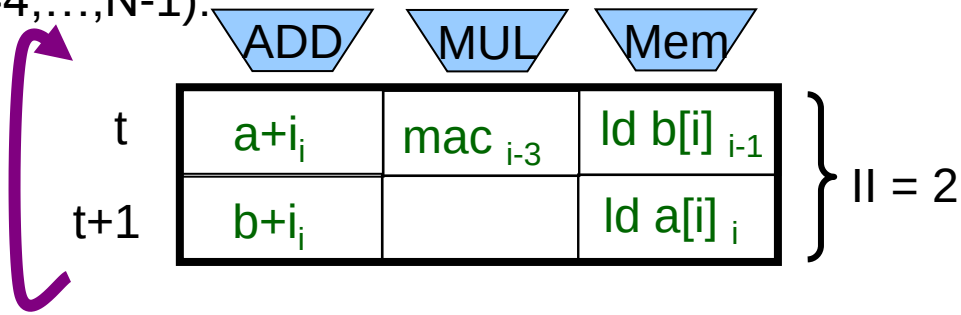
Simplified IR:



(loop control omitted for simplicity, alternatively, ZOL)

Given:
VLIW-Processor with 3 units:
- Adder (Latency 1),
- Multiplier/MAC (Latency 3),
- Memory access unit (Latency 3)

Kernel
(i=4,…,N-1):



Summary
Software Pipelining / Modulo-Scheduling

- Software Pipelining: Move operations across iteration boundaries
  - Simplest technique: Modulo scheduling
    = Fill modulo reservation table
- ☺ Better resource utilization,
  more ILP,
  also in the presence of loop-carried data dependences
- ☹ In general, higher register need, maybe longer code
- Heuristics e.g. HRMS, Swing Modulo Scheduling, …
- Optimal methods e.g. Integer Linear Programming
  - (Problem is NP-complete like acyclic scheduling)
- Self-overlapping live ranges need special treatment
- Loop unrolling can leverage additional optimization potential
- Up to now, only at target code level,
  hardly integrated (sometimes with register allocation)