# Instruction-Level Parallel Processor Architectures

# Instruction Scheduling

**Local and Global Scheduling**

Christoph Kessler, IDA,
Linköpings universitet, 2014.



# RISC and Instruction-Level Parallel Target Architectures

Christoph Kessler, IDA,
Linköpings universitet, 2014.

## CISC vs. RISC

**CISC**

- Complex Instruction Set Computer
- Memory operands for arithmetic and logical operations possible
- M(r1+r2) ← M(r1+r2) * M(r3+disp)
- Many instructions
- Complex instructions
- Few registers, not symmetric
- Variable instruction size
- Instruction decoding (often done in microcode) takes much silicon overhead
- Example: 80x86, 680x0

**RISC**

- Reduced Instruction Set Computer
- Arithmetic/logical operations only on registers
- add r1, r2, r1
  load r1, r4
  load r3+disp, r5
  mul r4, r5
  store r5, r1
- Few, simple instructions
- Many registers, all general-purpose typ. 32 ... 256
- Fixed instruction size and format
- Instruction decoding hardwired
- Example: POWER, HP-PA RISC, MIPS, ARM, SPARC



## Instruction-Level Parallel (ILP) architectures

**Single-Issue:** (can start at most one instruction per clock cycle)

- Simple, pipelined RISC processors with one or multiple functional units
  - e.g. ARM, DLX

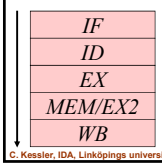
**Multiple-Issue:** (can start several instructions per clock cycle)

- Superscalar processors
  - e.g. Sun SPARC, MIPS R10K, Alpha 21264, IBM Power2, Pentium
- VLIW processors
  - e.g. Multiflow Trace, Cydrome Cydra-5, Intel i860, HP Lx, Transmeta Crusoe; most DSPs, e.g. Philips Trimedia TM32, TI 'C6x
- EPIC processors
  - e.g. Intel Itanium family (IA-64)



## Pipelined RISC Architectures

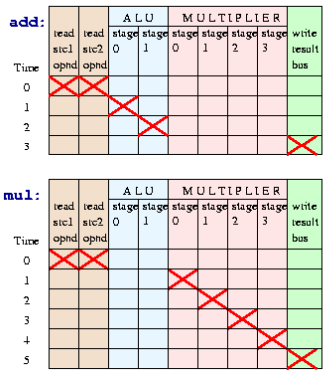
- A single instruction is issued per clock cycle
- Possibly several parallel functional units / resources
- Execution of different phases of subsequent instructions overlaps in time. This makes them prone to:
  - **data hazards** (may have to delay op until operands ready),
  - **control hazards** (may need to flush pipeline after wrongly predicted branch),
  - **structural hazards** (required resource(s) must not be occupied)
- Static scheduling (insert NOPs to avoid hazards)
  vs. Run-time treatment by automatic hazard detection + pipeline stalling




| issue | cycle | $PM$ | Decoder | $ALU_1$ | $DM/ALU_2$ | Regs |
|---|---|---|---|---|---|---|
| $I_1$ | 1 | $IF_1$ | | | | |
| $I_2$ | 2 | $IF_2$ | $ID_1$ | | | |
| $I_3$ | 3 | $IF_3$ | $ID_2$ | $EX_1$ | | |
| $I_4$ | 4 | $IF_4$ | $ID_3$ | $EX_2$ | $MEM_1$ | |
| $I_5$ | 5 | $IF_5$ | $ID_4$ | $EX_3$ | $MEM_2$ | $WB_1$ |
| $I_6$ | 6 | $IF_6$ | $ID_5$ | $EX_4$ | $MEM_3$ | $WB_2$ |



## Reservation Table, Scheduling Hazards




**Reservation table** specifies required resource occupations

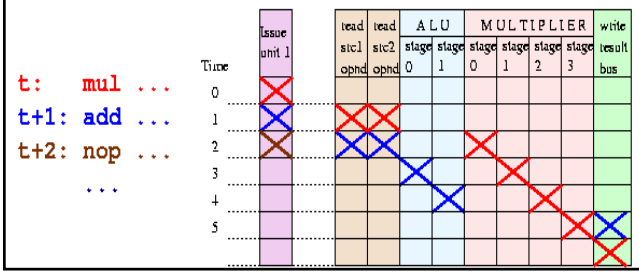
[Davidson 1975]

```
t:    mul ...
t+1:  ...
t+2:  add ...
...
```

*structural hazard at t=5*

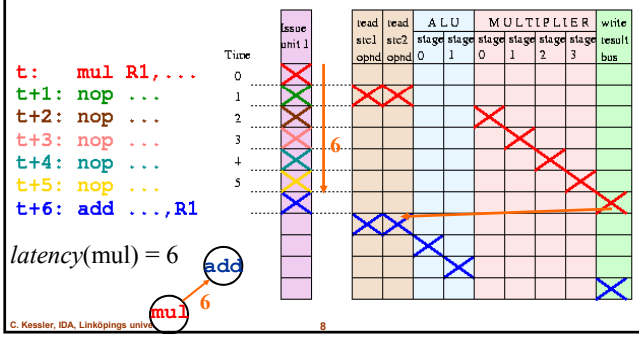
## Instruction Scheduling (1)

- Map instructions to time slots on issue units (and resources), such that no hazards occur
  → ***Global reservation table, resource usage map***




## Instruction Scheduling (2)

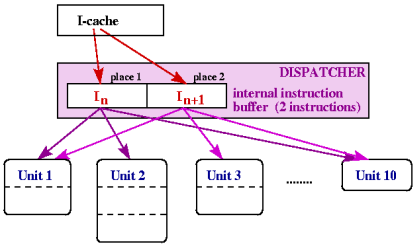
- Data dependences imply **latency constraints**
  → target-level data flow graph / data dependence graph






## Superscalar processor

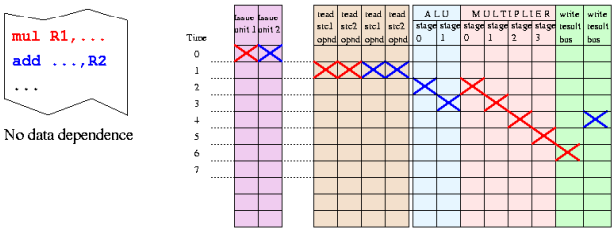
- Run-time scheduling by instruction dispatcher
  - convenient (sequential instruction stream – as usual)
  - limited look-ahead buffer to analyze dependences, reorder instr.
  - high silicon overhead, high energy consumption
- Example: Motorola MC 88110
  2-way, in-order issue superscalar






## Dual-Issue (w=2)

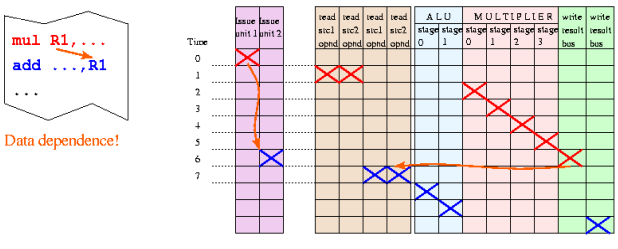
- Example (1):
  Linear code "mul R1,…; add …,R2" expands to






## Dual-Issue (w=2)

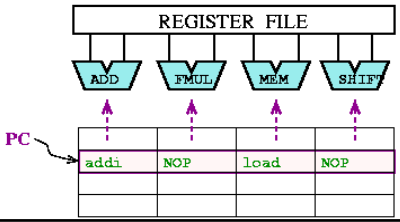
- Example (2):
  Linear code "mul R1,…; add …,R1" expands to





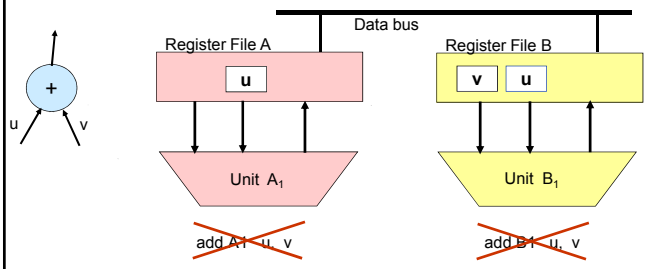

## VLIW (Very Long Instruction Word) architectures

- Multiple slots for instructions in long instruction-word
  - Direct control of functional units and resources – low decoding OH
- Compiler (or assembler-level programmer) must determine the schedule statically
  - independence, unit availability, packing into long instruction words
  - Challenging! But the compiler has more information on the program than an on-line scheduler with a limited lookahead window.
  - Silicon- and energy-efficient



## Clustered VLIW processor

- E.g., TI C62x, C64x DSP processors
- Register classes
- Parallel execution constrained by operand residence





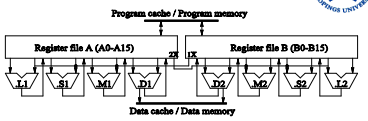

## EPIC architectures

- Based on VLIW
- Compiler groups instructions to LIW's (**bundles**, fetch units)
- Compiler takes care of resource and latency constraints
- Compiler marks sequences of independent instructions as **instruction groups** by inserting delimiters (**stop bits**)
- Dynamic scheduler assigns resources and reloads new bundles as required



## EPIC Example: Instruction format for TI 'C62x

- Texas Instruments DSP processor series TMS320C62xx




- 1 **fetch packet** (a very long instruction word) has 8 slots
  - may contain up to 8 **instruction groups** (**issue packets**) to be executed in sequence
  - Instruction groups are marked by chaining bits.
    - Up to 8 instructions in an instruction group
  - Instructions within an instruction group must use disjoint resources (basically, different functional units)
- Example: 3 issue groups { A||B||C } ; { D||E||F } ; { G||H }

| A | 1 | B | 1 | C | 0 | D | 1 | E | 1 | F | 0 | G | 1 | H | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|



## EPIC Example: Intel IA-64 (Itanium)

- Constraints on bundle contents and placement of delimiters for instruction groups: 24 templates

| | | | | | |
|---|---|---|---|---|---|
| M I I | M M I | M L I | M M F | M B B | M M B |
| M I I | M M I | M L I | M M F | M B B | M M B |
| M I I | M M I | M F I | M I B | B B B | M F B |
| M I I | M M I | M F I | M I B | B B B | M F B |

**Instruction types:**

M = Memory (on M unit)
I = Integer (on I unit)
F = Floatingpoint (on F unit)
B = Branch (on B unit)
A = supertype of I and M
LX = uses 2 bundle slots, uses I and B units
NOP can replace anything, uses no unit

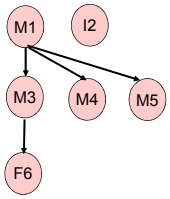
**Functional units:**


An implementation of the IA-64 instruction set interface is the Itanium processor series.



## Example: Local scheduling for IA-64

- A DAG with a greedy and an optimal schedule




| | | |
|---|---|---|
| M1 | I2 | NOP |
| M3 | M4 | NOP |
| M5 | F6 | NOP |

used templates:
MI;I / MMI; / MFI;

| | | |
|---|---|---|
| M1 | M3 | I2 |
| M4 | M5 | F6 |

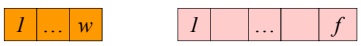
used templates:
M;MI / MMF;

Adapted from: S. Haga, R. Barua: EPIC Instruction Scheduling based on Optimal Approaches.



## A Generic ILP-Architecture Model for Retargetable Code Optimization

- Issue width *w*
  *w*-way in-order superscalar or size of longest instruction group
- *w* issue units
- *f* resources
  functional units, internal buses, …
- Instruction set *I*
  for each instruction *y* in *I*, specify its
  - syntax: mnemonic, parameters, types
  - semantics: (tree)pattern in terms of IR operations, latency
  - resource requirements: reservation table, issue unit(s)
- Formal specification in xADML [Bednarski'06]
  (register sets etc. not considered here)



DF00100 Advanced Compiler Construction
TDDC86 Compiler Optimizations and Code Generation

# Instruction Scheduling

## Overview

Christoph Kessler, IDA,
Linköpings universitet, 2014.

## Instruction Scheduling

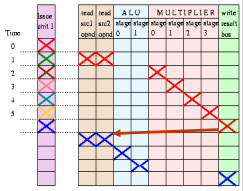
**Generic Resource model:** Reservation table

**Optimize:** time, space, energy

**Local Scheduling**
(f. Basic blocks / DAGs)

- Data dependences
  → Topological sorting
  - List Scheduling (diverse heuristics)
- Optimal Scheduling
  (Exhaustive search, DP, B&B, CLP, ILP)

```
t:   mul R1,...
t+1: nop ...
t+2: nop ...
t+3: nop ...
t+4: nop ...
t+5: nop ...
t+6: add ...,R1
```




**Global Scheduling**

- Code motion, Branch delay slot filling
- Trace scheduling, Region scheduling, ...
- Cyclic scheduling for loops (Software pipelining)

There exist **retargetable schedulers**
and **scheduler generators**, e.g. for GCC since 2003



DF00100 Advanced Compiler Construction
TDDC86 Compiler Optimizations and Code Generation

# Local Instruction Scheduling

Christoph Kessler, IDA,
Linköpings universitet, 2014.

## Optimization problems in local scheduling

MRIS – minimum register need instruction scheduling
+ Spilling (store/reload) takes additional time
+ Power consumption in embedded procs. increases with # mem. accesses
+ Superscalar processors with shadow registers and register renaming
  → compiler-generated spill code cannot be eliminated at run time
– NP–complete [Sethi'75]

MTIS – minimum time instruction scheduling
+ hiding pipeline delays
+ exploiting instruction–level parallelism (for superscalar/VLIW)
– NP–complete [Garey/Johnson'79, Gross'83, Lawler et al.'87]

RCMTIS – register–constrained minimum time instruction scheduling

SMRTIS – simultaneous minimization of space and time



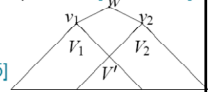
## MRIS: Space-optimal scheduling

(a) based on postorder traversal of the DAG

Special case: tree: space-opt. schedule in linear time [Sethi, Ullman '70]
Special case: vector tree (node size attribute): space-opt. $O(n \log n)$ [Rauber'90]

Special case: series-par. DAG: space-opt. schedule in pol. time [Güttler '81]

General DAG, contiguous schedules ($\leq 2^n$)
- Random dfs [K., Paul, Rauber '91]
- Enumeration with DC strategy [K., Rauber '93/'95]




(b) based on topological sorting of the DAG → general schedules ($\leq n!$)
- space-optimal (enumeration + dynamic programming) [K. '96]

(c) based on finding instruction lineages in the DAG
- heuristic method by [Govindarajan et al. '00]

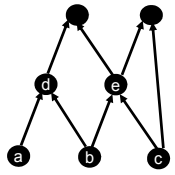
→ see separate lecture on MRIS



## Example: Topological Sorting (0)

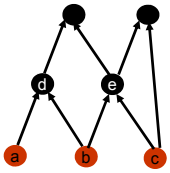
- Not yet considered
- Data ready (zero-indegree set)
- Already scheduled, still alive
- Already scheduled, no longer referenced

Given:
Data flow graph of a basic block
(a directed acyclic graph, DAG)



## Example: Topological Sorting (1)

- Not yet considered
- Data ready (zero-indegree set)
- Already scheduled, still alive
- Already scheduled, no longer referenced

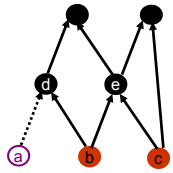



a



## Example: Topological Sorting (2)

- Not yet considered
- Data ready (zero-indegree set)
- Already scheduled, still alive
- Already scheduled, no longer referenced

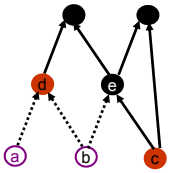



a b



## Example: Topological Sorting (3)

- Not yet considered
- Data ready (zero-indegree set)
- Already scheduled, still alive
- Already scheduled, no longer referenced

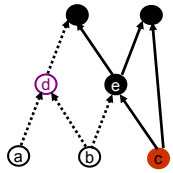



a b d



## Example: Topological Sorting (4)

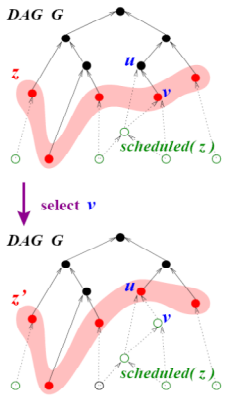
- Not yet considered
- Data ready (zero-indegree set)
- Already scheduled, still alive
- Already scheduled, no longer referenced




a b d and so on...



## List scheduling = Topological sorting




*top_sort*( Set $z$, int[] $INDEG$, int $t$ )
if $z \neq \emptyset$ // $(t \leq n)$
  select arbitrary node $v \in z$;
  // *implicitly remove all edges* $(v,u)$ $\forall u$:
  $INDEG'(u) = \begin{cases} INDEG(u)-1 & \text{where } \exists (v,u) \\ INDEG(u) & \text{elsewhere} \end{cases}$
  // *update zero-indegree set:*
  $z' \leftarrow z - \{v\} \cup \{\text{new leaves}\}$
  $= \{u : INDEG(u) = 0\}$
  $S[t] \leftarrow v$;
  *top_sort*( $z'$, $INDEG'$, $t+1$ );
else output $S[1:n]$ fi

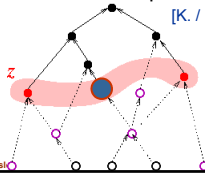
Call *top_sort*( $z_0$, $INDEG_0$, 1 )
produces a schedule in $S[1:n]$

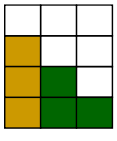


## Topological Sorting and Scheduling

- Construct schedule incrementally in topological (= causal) order
  - "Appending" instructions to partial code sequence: close up in target schedule reservation table (as in "Tetris")
  - Idea: Find optimal target-schedule by enumerating all topological sortings ...
    - Beware of scheduling anomalies with complex reservation tables!

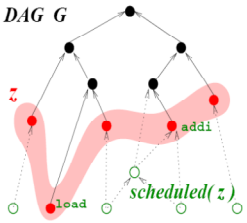
[K. / Bednarski / Eriksson 2007]

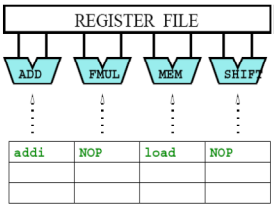


## Greedy List Scheduling for VLIW (1)

A greedy heuristic for list scheduling
fills in one step as many slots in a VLIW word as possible
with ready instructions of the zeroindegree set.

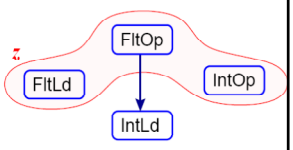





## Greedy List Scheduling for VLIW (2)

A greedy heuristic for list scheduling
fills in one step as many slots in a VLIW word as possible
with ready instructions of the zeroindegree set.




Not optimal!

(optimal) greedy schedule:

| t | IntFlt-Unit | IntMem-Unit |
|---|---|---|
| 1 | FltOp | FltLd |
| 2 | IntOp | IntLd |

(non-optimal) greedy schedule:

| t | IntFlt-Unit | IntMem-Unit |
|---|---|---|
| 1 | FltOp | IntOp |
| 2 | — | IntLd |
| 3 | — | FltLd |



## Local Scheduling Heuristics

- **List Scheduling Heuristics**
  - **Deepest Level First** (a.k.a. highest level first etc.)
    - Select, among ready instructions, one with longest accumulated latency on a path towards any dependence sink (root node)
    - Forward vs Backward scheduling
- **Critical Path Scheduling**
  - Detect a *critical path* (longest accumulated latency) in the DAG, schedule its nodes → partial schedule, and remove them from the DAG.
  - Repeat until DAG is empty, splicing in new nodes between scheduled ones as appropriate, or inserting fresh time slots where needed



DF00100 Advanced Compiler Construction
TDDC86 Compiler Optimizations and Code Generation

# Global Instruction Scheduling

Christoph Kessler, IDA,
Linköpings universitet, 2014.

## Scheduling Branch Instructions

- **Delayed Branch**
  - Effect of conditional branch on program counter is delayed
  - 1 or more instructions after a branch instruction are always executed, independent of the condition outcome
    - SPARC, HP PA-RISC: 1 delay slot
    - TI 'C62x: 5 delay slots
- Scheduling: Fill delay slots with useful instruction if there is one, otherwise with NOP
- Heuristic for finding candidate instructions:
  1. Instructions from same basic block that are not control dependent on the branch and that the condition is not data dependent of
  2. Instructions from most likely branch target basic block for speculative execution
  - See e.g. [Muchnick Ch. 17.1.1] for further details



## Remark: Hardware Support for Branch Delay Slot Filling

Nullification feature (SPARC) for conditional branch delay slots in one branch direction

- activated by setting a bit in the instruction opcode
- delay slot instruction executed only if a conditional branch is taken
  treated as `NOP` if not taken.
- not applicable to "branch always": fixed delay slot

## Trace Scheduling

developed for VLIW architectures [Fisher'81] [Ellis'85]

- idea: enlarge the scope of local scheduling to traces

  trace = acyclic path of basic blocks in the CFG

  track execution frequencies for BB's/traces (e.g., profiling)

- idea: make the most frequent trace fast:

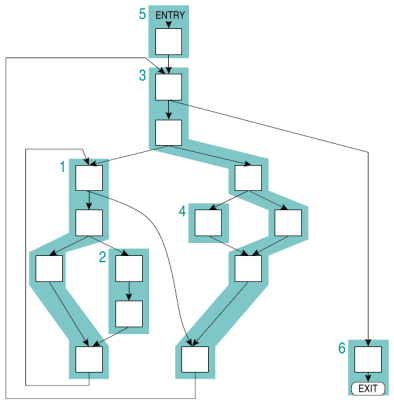
  + virtually merge BB's in the most frequent trace
    schedule trace as one BB, e.g. by greedy VLIW list scheduling

  + insert compensation code in less frequent side traces for correctness
    → accept slowdown for side traces
    → program lenght may grow (worst case: exponentially)

  + continue same procedure with next frequent trace



## Trace Scheduling (2)




Traces in a control flow graph,

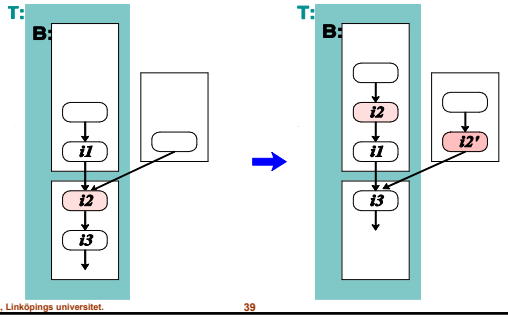
numbered in order of decreasing execution frequency

A trace ends at a backward branch, or at a join point with another trace of higher execution frequency (which thus was constructed earlier).



## Trace Scheduling (3)

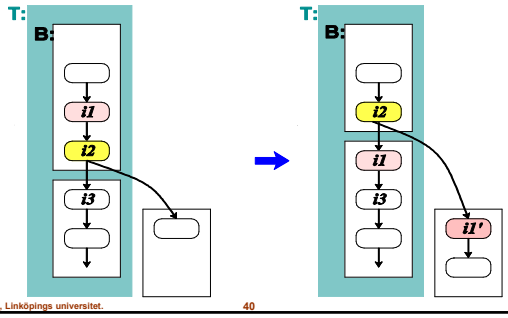
- Insertion of compensation code
  - **Case:** When moving an instruction *i2* to a predecessor block *B* in the trace *T* (e.g., to fill a branch delay slot)






## Trace Scheduling (4)

- Insertion of compensation code
  - **Case:** When moving an instruction *i1* to a successor block of *B* in the trace *T*






## Trace Scheduling (5)

- **Summary of cases:**

  Code reordering with insertion of compensation code
  Hoisting an assignment
  Interchange assignment and label
  Moving assignments across conditional branches
  Moving a branch
  interchange branches



## Region Scheduling

[Gupta/Soffa'90]

Idea: avoid idle cycles caused by regions with insufficient parallelism

Program region = one or several BB's that require the same control conditior

Repeatedly apply a set of local code transformations:

- loop unrolling
- moving instructions from BB's with excessive parallelism into BB's with insufficient parallelism
- merging of regions

to balance the degree of parallelism

Heuristic measure for average degree of parallelism in a region:
# instructions(region) / length of critical path(region)

## Program regions for global scheduling

- **Trace** (see above)
  - A path of basic blocks
- **Superblock**
  - A trace with the restriction that there may be no branches into any of its basic blocks, except the first one
- **Treegions = Extended Basic Blocks**
  - An out-tree of basic blocks – no branch into any of its basic blocks, except the first one
- **Hyperblock**
  - A single-entry, multiple exit region with internal control flow. As superblocks, but allow *hammocks* resolved by predication.
- All these regions are acyclic (but may be part of a cycle around)
- Traces and superblocks are "linear regions", while treegions and hyperblocks are "nonlinear regions"



## Summary + Outlook: Instruction Scheduling

- usually, optimize for time (other important metrics: space, energy)
  → see also lecture on energy-aware code generation
- local methods
  postorder traversals, forward/backward list scheduling, optimal methods
  → see also lecture on space-optimal scheduling (MRIS)
- global methods
  trace scheduling, percolation scheduling, region scheduling
  → see also lecture on software pipelining
- interferences with instruction selection, register allocation,
  → phase-ordering problems
  → see also lecture on integrated code generation
- interferences with data layout, exploit advanced addressing units, ...
  → see also lecture on code generation for DSPs



## Further scheduling issues, not covered

- creating and scheduling predicated code
- speculation (with and without hardware support)
  prefetching (load speculation), branch speculation, value speculation ...
- run-time scheduling, profile-driven scheduling
- automatic generation of instruction schedulers: finite state automata
  [Proebsting/Fraser: *Detecting Pipeline Hazards Quickly*, POPL'94],
  [Bala/Rubin MICRO-28, 1995]
  e.g. the new GCC scheduler [Makarov, GCC Dev. Summit 2003]



## Generation of Instruction Schedulers

- Given: Instruction set with
  - reservation table for each instruction
- Set of resource-valid schedules = regular language over the alphabet of instructions
- Scheduling instr. A after B leads to a certain **pipeline state** (functional unit reservations and pending latencies of recently issued instructions)
- Scheduling A in pipeline state q leads to new pipeline state q'
- → **Finite automaton** ("Müller automaton") of all possible pipeline states and (appending) scheduling transitions
  - Or finite transducer → gives also the time offset for next instruction
- Precompute possible states + transitions → Scheduling much faster (table lookup instead of interpreting reservation table composition)
- Reversed automaton to allow insertions at any location
- Automata become huge! But can be optimized.



## Recommended Reading (global scheduling)


- J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Computers*, **30**(7):478–490, 1981.
- Paolo Faraboschi, Joseph A. Fisher, Cliff Young: Instruction Scheduling for Instruction Level Parallel Processors. *Proceedings of the IEEE*, vol. **89** no. 11, Nov. 2001
- Daniel Kästner, Sebastian Winkel: ILP-based Instruction Scheduling for IA-64. *Proc. ACM SIGPLAN LCTES-2001*, June 2001
- Sebastian Winkel. *Optimal Global Instruction Scheduling for the Itanium® Processor Architecture*. Ph.D. thesis. Saarland University, Saarbrücken, Germany, 2004. ISBN 3-937436-01-6




## Recommended Reading (Generating Schedulers from Reservation Tables)


- T. Müller: Employing finite automata for resource scheduling. Proc. MICRO-26, 1993
- Proebsting, Fraser: Detecting pipeline structural hazards quickly. Proc. ACM POPL-1994
- Bala, Rubin: Efficient instruction scheduling using finite state automata. Proc. MICRO-28, 1995
- Eichenberger, Davidson: A reduced multi-pipeline machine description that preserves scheduling constraints. Proc. ACM PLDI-1996