Instruction-Level Parallel
Processor Architectures

Instruction Scheduling
Local and Global Scheduling

CISC vs. RISC
CISC
- Complex Instruction Set Computer
- Memory operands for arithmetic and logical operations possible
- $M(r_1+r_2) \neq M(r_1)+M(r_2)$
- 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
- $r_1 = r_1 + r_2$
- Few, simple instructions
- Many registers, all general-purpose
- Fixed instruction size and format
- Instruction decoding hardwired
- Example: POWER, HP-PA RISC, MIPS, ARM, SPARC

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

Reservation Table, Scheduling Hazards

Reservation table
specifies required resource occupations

[Davidson 1975]
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

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

VLIW (Very Long Instruction Word) architecture

- 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 look-ahead window.
  - Silicon- and energy-efficient
### Clustered VLIW processor
- E.g., TI C62x, C64x DSP processors
- Register classes
- Parallel execution constrained by operand residence

![Clustered VLIW processor diagram](image)

### 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
- 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 (All/I/C) ; (D/I/E/F) ; (G/I/H)

### EPIC Example: Intel IA-64 (Itanium)
- Constraints on bundle contents and placement of delimiters for instruction groups: 24 templates

### Example: Local scheduling for IA-64
- A DAG with a greedy and an optimal schedule

### 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
  - \( i \) resources
  - 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)
Instruction Scheduling

Overview

Generic Resource model: Reservation table
Optimize: time, space, energy

Local Scheduling
- (I. Basic blocks / DAGs)
  - Data dependences
  - Topological sorting
  - List Scheduling (diverse heuristics)
  - Optimal Scheduling
    - Exhaustive search, DP, B&B, CLP, ILP

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

Optimization problems in local scheduling
- Spilling (store/reload) takes additional time
- Power consumption in embedded proc. increases with # mem. accesses
- Superscalar processors with shadow registers and register renaming
  - Compiler-generated spill code cannot be eliminated at run time
  - NP-complete

MRIS – minimum register need instruction scheduling
- (a) based on postorder traversal of the DAG
  - Special case: tree, space-opt. schedule in linear time [Sethi, Ullman ‘79]
  - Special case: vector tree (node size attribute); space-opt. O(n log n) [Rauber’89]
  - General DAG, contiguous schedules (≤ 2)
    - Random ods [K. Paul, Rauber ‘91]
    - Enumeration with DC strategy [K., Rauber ‘93/95]

(b) based on topological sorting of the DAG → general schedules (≤ n)
  - space-optimal (enumeration + dynamic programming) [K. ‘90]

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

MRIS: Space-optimal scheduling

Example: Topological Sorting (0)

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

**Example: Topological Sorting (2)**

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

**Example: Topological Sorting (3)**

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

**Example: Topological Sorting (4)**

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

**List scheduling = Topological sorting**

```
top_sort( Set S, int INDEG )
if IS S
select arbitrary node v \in S
// implicitly remove all edges (v, w) \forall w:
INDEG(w) = \begin{cases} 
INDEG(w) - 1 & \text{where } v \in S \\
INDEG(w) & \text{elsewhere} 
\end{cases}
// update zero-indegree set:
w' = \{ w \in S \} \cup \{ new leaves \}
= \{ w : \text{INDEG}(w) = 0 \}
S' = v
Call top_sort( S', INDEG', +1 )
else output S : \forall i \in S 
```

**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 zerodegree 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 zerodegree set.

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

Global Instruction Scheduling

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

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 length 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 \( i_2 \) 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 \( i_1 \) 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 condition

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:

\[ \text{# instructions}(\text{region}) / \text{length of critical path}(\text{region}) \]
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

- **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
  - 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.

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]

Recommended Reading (global scheduling)

- Paolo Faraboschi, Joseph A. Fisher, Cliff Young: Instruction Scheduling for Instruction Level Parallel Processors: *Proceedings of the IEEE*, vol. 89 no. 11, Nov. 2001

Recommended Reading (Generating Schedulers from Reservation Tables)

- Proebsting, Fraser: Detecting pipeline structural hazards quickly. *Proc. ACM POPL-1994*