CISC vs. RISC

CISC
- Complex Instruction Set Computer
- Memory operands for arithmetic and logical operations possible
  \[ M(r1+r2) \leftarrow M(r1+r2) \cdot M(r3+disp) \]
- 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
- \( \text{add } r1, r2, r1 \)
- \( \text{load } r1, r4 \)
- \( \text{load } r3+\text{disp}, r5 \)
- \( \text{mul } r4, r5 \)
- \( \text{store } r5, r1 \)
- 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, Cyoem Cydra-5, Intel i860, HP Lx, Transmeta Crusoe; most DSPs, e.g. Philips Trimedia TM32, TI 'C6x
- EPIC processors
  - e.g. Itallian 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

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

**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 }
Instruction Scheduling

Overview

Local Instruction Scheduling

Optimization problems in local scheduling

MRIS: Space-optimal scheduling

Example: Topological Sorting (0)
**Example: Topological Sorting (1)**

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

![Diagram](image1)

**Example: Topological Sorting (2)**

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

![Diagram](image2)

**Example: Topological Sorting (3)**

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

![Diagram](image3)

**Example: Topological Sorting (4)**

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

![Diagram](image4)

**List scheduling = Topological sorting**

- `d = G, n = 0`
- `t = 1`
- `top_sort( Set z, int[] INDEG, int t )`
- `if z ≠ ∅`
  - `select arbitrary node v ∈ z`
  - `INDEG = { INDEG(v) - 1 where 3(v,a) ∈ INDEG }, elsewhere`
  - `z = z ∪ {v}`
  - `top_sort( z, INDEG, t + 1 )`
- `else output S[1..n]`
- `Call top_sort( ∅, INDEG, 1 )`
- `produces a schedule 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
  - 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.

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 \(\rightarrow\) 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
  - Heuristic for finding candidate instructions:
    1. Instructions from the 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 length may grow (worst case: exponentially)
  • continue same procedure with next frequent trace

Trace Scheduling (2)

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/Sofa’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
  • merging of regions
  to balance the degree of parallelism

Heuristic measure for average degree of parallelism in a region:
$$\frac{\# \text{ instructions(region)}}{\text{length of critical path(region)}}$$
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
- **Tree-Regions = 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) and traces and superblocks are "linear regions", while tree-regions 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 traversal, forward/backward list scheduling, optimal methods
  → see also lecture on space-optimal scheduling (MHTS)
- 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 65K

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)

- 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,
  University, Saarbrücken, Germany, 2004. ISBN 3-937436-01-6

Recommended Reading (Generating Schedulers from Reservation Tables)

- Proebsting, Fraser: Detecting pipeline structural hazards quickly. Proc. ACM POPL-1994
- Eichenberger, Davidson: A reduced multi-pipeline machine description that preserves scheduling constraints. Proc. ACM
  PLDI-1996