1. RISC and Instruction-Level Parallel Target Architectures

### CISC vs. RISC

**CISC**
- Complex Instruction Set Computer
- Memory operands for arithmetic and logical operations possible
- \( M(r1+r2) \), \( M(r1+r2) * M(r3+\text{disp}) \)
- 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
- Addition, load, store, multiply
- Few, simple instructions
- Many registers, all general-purpose
- Fixed instruction size and format
- Instruction decoding hardwired in microcode
- 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 cp 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 pipeline stallings

### Reservation Table, Scheduling Hazards

Reservation table specifies required resource occupations
[Davidson 1975]
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

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

EPIC architectures

- Based on VLIW
- Compiler groups instructions to LIW’s (bundles)
- Compiler takes care of resource and latency constraints
- Compiler marks sequences of independent instructions
- Dynamic scheduler assigns resources and reloads new bundles as required
Instruction Scheduling

Generic Resource model
- Reservation table

Local Scheduling
- (f. Basic blocks / DAGs)
- Data dependences
- Topological sorting
- List Scheduling
  - (diverse heuristics)

Global Scheduling
- Trace scheduling, Region scheduling, ...
- Cyclic scheduling (Software pipelining)

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

Example: Topological Sorting (0)
- Not yet considered
- Data ready (zero-indegree set)
- Already scheduled, still alive
- Already scheduled, no longer referenced

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

and so on...
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]

Software Pipelining of Loops (1)

Software Pipelining of Loops (2)

Software Pipelining of Loops (3)

Register Allocation

- **Register Allocation:** Determines values (variables, temporaries, constants) to be kept when in registers
- **Register Assignment:** Determine in which physical register such a value should reside.
  - Essential for Load-Store Architectures
  - Reduce memory traffic (⇒ memory / cache latency, energy)
  - Limited resource
  - Values that are alive simultaneously cannot be kept in the same register
  - Strong interdependence with instruction scheduling
    - scheduling determines live ranges
    - spill code needs to be scheduled
  - **Local register allocation** (for a single basic block) can be done in linear time (see function getreg() above).
  - **Global register allocation** (with minimal spill code) is NP-complete. Can be modeled as a graph coloring problem [Ershov'62] [Cocke'71].
Local Register Allocation

For variable \( v \) and basic block \( B \):

\[ \text{netsave}(v, B) = \text{netsave} + \text{defs} - \text{ddefs} \]

- \( b \)-cost \( (i = 1 \text{ if } \text{Load}(v) \text{ needed at beg. of } B, 0 \text{ otherwise}) \)
- \( s \)-cost \( (s = 1 \text{ if } \text{Store}(v) \text{ needed at end of } B, 0 \text{ otherwise}) \)

For loop \( l \), estimate benefit of keeping \( v \) in a register:

\[ \text{benefit}(v, l) = 10^{b\text{-cost}} \cdot \sum_{i \in \text{ddefs}(i)} \text{netsave}(v, i) \]

with \( R \) registers available:

- allocate the \( R \) objects with greatest benefit in \( l \).
- moves may be necessary instead of \( \text{Load}(v) / \text{Store}(v) \)
- if \( v \) could reside in (different) registers in \( \text{Pred}(l), B, \text{Succ}(l) \)
- add worst-case terms \( |\text{Pred}(v)| \cdot \text{mcost}, |\text{succ}(v)| \cdot \text{mcost} \)

Live range

(Here, variable = program variable or temporary)

- A variable is being defined at a program point if it is written (given a value) there.
- A variable is used at a program point if it is read (referenced in an expression) there.
- A variable is alive at a point if it is referenced there or at some following point that has not (may not have) been preceded by any definition.
- A variable is reaching a point if an (arbitrary) definition of it, or usage (because a variable can be used before it is defined) reaches the point.
- A variable’s live range is the area of code (set of instructions) where the variable is both alive and reaching.
- does not need to be consecutive in program text.

Register Allocation for Loops

Example:

\[
\begin{align*}
x_3 &= 7 \\
\text{for } i = 1 \text{ to } 100 \{ \\
x_1 &= x_3 + 2 \\
x_2 &= x_1 + x_3 \\
x_3 &= x_2 + x_1 \\
y &= x_3 + 42 \\
\}
\]

Live ranges (loop only): cyclic intervals e.g. for \( i : [0, 6), (6, 7) \)

Control flow graph

Register Allocation by Graph Coloring

Step 1: Given a program with symbolic registers \( s_1, s_2, ... \)

- Determine live ranges of all variables

\[
\begin{align*}
i &= c+4; \text{load } @(fp), s_1 & c \\
\text{add } s_1, s_4, s_2 & c \\
\text{store } s_2, 4@(fp) & i \\
d &= c-2; \\
\text{sub } s_1, s_2, s_3 & d \\
\text{store } s_3, 12@(fp) & !d \\
c &= c+1; \text{mul } s_1, s_2, s_4 & c \\
\text{store } s_4, 8 @(fp) & !c \\
\end{align*}
\]

Step 2: Build the Register Interference Graph

- Undirected edge connects two symbolic registers \( (s_i, s_j) \) if live ranges of \( s_i \) and \( s_j \) overlap in time
- Reserved registers (e.g. fp) interfere with all \( s_i \)

Step 3: Color the register interference graph with \( k \) colors, where \( k = \# \text{available registers} \)

- If not possible: pick a victim \( s_i \) to spill, generate spill code (store after def., reload before use)
- This may remove some interferences.

Register Allocation by Graph Coloring

- Input: a program and available registers
- Output: a register allocation

Step 3: Color the register interference graph with \( k \) colors, where \( k = \# \text{available registers} \)

- If not possible: pick a victim \( s_i \) to spill, generate spill code (store after def., reload before use)
- This may remove some interferences.

Rebuild the register interference graph + repeat Step 3...

This register interference graph cannot be colored with less than 4 colors, as it contains a 4-clique
Coloring a graph with \( k \) colors

- NP-complete for \( k > 3 \)
- Chromatic number \( \chi(G) \) = minimum number of colors to color a graph \( G \)
- \( \chi(G) \rightarrow c \) if the graph contains a \( c \)-clique
  - A \( c \)-clique is a completely connected subgraph of \( c \) nodes

Chaitin’s heuristic (1981):

\[
S \leftarrow \{ s_1, s_2, \ldots \} \quad // \text{set of spill candidates} \\
\text{while } (S \text{ not empty}) \\
\quad \text{choose some } s \in S: \\
\quad \text{if } s \text{ has less than } k \text{ neighbors in the graph} \\
\quad \quad \text{then } // \text{there will be some color left for } s: \\
\quad \quad \quad \text{delete } s \text{ (and incident edges) from the graph} \\
\quad \quad \text{else } \text{modify the graph (spill, spill, coalesce ... nodes)} \\
\quad \quad \text{and restart} \\
\quad \quad \text{// once we arrive here, the graph is empty:} \\
\quad \quad \quad \text{color the nodes greedily in reverse order of removal.}
\]

Live range splitting

- Instead of spilling completely (reload before each use), it may be sufficient to split a live range at one position where register pressure is highest
  - save, and reload once

Chaitin’s Register Allocator (1981)

- For a copy instruction \( s_j \leftarrow s_i \)
  - where \( s_i \) and \( s_j \) do not interfere
  - and \( s_i \) and \( s_j \) are not rewritten after the copy operation
- Merge \( s_i \) and \( s_j \):
  - patch (rename) all occurrences of \( s_i \) to \( s_j \)
  - update the register interference graph
- and remove the copy operation.

Live range coalescing

- Instead of spilling completely (reload before each use), it may be sufficient to split a live range at one position where register pressure is highest

Phase ordering problems
Phase ordering problems (1)

Instruction scheduling vs. register allocation

(a) Scheduling first:
- determines Live-Ranges
- Register needs
- possibly spill-code to be inserted afterwards

(b) Register allocation first:
- Reuse of same register by different values introduces "artificial" data dependencies
- Constraints scheduler

Phase ordering problems (2)

Conflicts Instruction selection ↔ Scheduling / Reg. alloc.

- Selection first: Cost attribute of a pattern covering rule is only a coarse estimate of the real cost (effect on e.g. overall time)
  - Real cost based on:
    - currently free functional units
    - other instructions ready to execute simultaneously
    - pending latencies of already issued but unfinished instructions
  - Integration with instruction scheduling desirable
- Mutations with different resource requirements:
  - $a = 2 \times b$ or $a = b \ll 1$ or $a = b + b$?
- Different instructions with different register needs

Clustered VLIW processor

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

More phase ordering problems

In parallel e.g.:
- load on A || load on B || move A → B
- Mapping instructions to cluster
- Should preferably know already beforehand about free move-slots in schedule...
- Instruction scheduling
- Must know mapping to generate moves where needed
- Heuristic [Leupers'00]
- Iterative optimization by simulated annealing

Integrated Code Generation

Related Work

- Heuristic Phase Interleaving [Goodman/Hsu '88], [Leupers'00], ...
- Combination of 2 phases
  - Instruction selection and register allocation
    - DP for space optimization, e.g. [Aho/Johnson '77]
    - DP for space-time optimization, e.g. [Fraser et al.'92]
  - Instruction scheduling and register allocation
    - ILP e.g. [Kästner'00]
- Combination of 3 phases
  - ILP [Wilson et al.'94]
  - Only for simple, non-pipelined RISC processor
Results (2) – DP

Clustered 8-issue VLIW processor TI C6201

Leupers’ example

Results (4) – Optimal integrated code generation, OPTIMIST – DP algorithm

Single-issue vs. Single-cluster vs. Double-cluster Architecture

Our project: OPTIMIST

Retargetable integrated code generator

Processor specification language xADMML

Project Literature (Selection)

- www.ida.liu.se/~chrke/optimist