1. RISC and Instruction-Level Parallel Target Architectures

CISC vs. RISC
- CISC: Complex Instruction Set Computer
  - 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
  - Few, simple instructions
  - Many registers, all general-purpose
  - 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. ARM9, 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 (Very Long Instruction Word)
    - e.g. Multiflow Trace, Cydrome Cydra-5, Intel i860, HP Lx, Transmeta Crusoe;
      most DSPs, e.g. Philips Trimedia TM32, TI 'C6x
  - EPIC processors (Explicitly Parallel Instruction Computing)
    - e.g. Intel Itanium family (IA-64)

Processors with/without Pipelining
- Traditional processor without pipelining
  - One instruction takes 4 processor cycles, i.e. 0.25 instructions/cycle

- Processor with Simple Pipelining
  - An instruction takes 1 cycle on average with pipeline
    - 1 instruction/cycle

  This pipeline achieves 4-way parallelism
Processor with Super-Pipelining

A new instruction can begin before the previous one is finished. Thus you manage on average 3 instr/cycle when the pipeline is full.

A Processor with Parallel Pipelines

Problems using Branch Instructions on Simple Pipelined Processors

Branch instructions force the pipeline to restart and thus reduce performance.

The diagram below shows execution of a branch (cbr = conditional branch) to instruction #3, which makes the pipeline restart.

The grey area indicates lost performance. Only 4 instructions start in 6 cycles instead of the maximum of 6.

Summary 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) e.g. functional units, bus, register, must not be occupied)
  - Static scheduling (insert NOPs to avoid hazards) vs. Run-time treatment by pipeline stalling

Reservation Table, Scheduling Hazards

(Avoid hazards = resource collisions)

Comparison between Superscalar Processors and VLIW processors

Superscalar Processors

with multiple loading of instructions (multi-issue)

VLIW Processors

(Very Long Instruction Word)

Several processor units are loaded simultaneously on different operations in the same instructions. E.g. the multiflow machine, 1024 bits, 28 operations, or specialized graphics processors
Superscalar Processors

A superscalar processor has several function units that can work in parallel and which can load more than 1 instruction per cycle. The word superscalar comes from the fact that the processor executes more than 1 instruction per cycle. 

The diagram below shows how a maximum of 4 units can work in parallel, which in theory means they work 4 times faster.

Branch Effects on Performance for Deeply Pipelined Superscalar Processors

Branch-instructions force the pipeline to restart and thus reduce performance. Worse on deeply pipelined superscalar processors.

VLIW (Very Long Instruction Word) architecture

- Multiple slots for instructions in long instruction-word
- 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.

EPIC Architectures

(Explicitly Parallel Instruction Computing)

- 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
2. Instruction Scheduling

The Instruction Scheduling Problem

- Schedule the instructions in such an order that parallel function units are used to the greatest possible degree.

Input:
- Instructions to be scheduled
- A data dependency graph
- A processor architecture
- Register allocation has (typically) been performed

Output:
- A scheduling of instructions which minimizes execution time

Example Instructions to be Scheduled

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Dependency graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld [%sp + 0x64], %g1</td>
<td></td>
</tr>
<tr>
<td>ld [%sp + 0x68], %l1</td>
<td></td>
</tr>
<tr>
<td>add %l4, %g1, %l2</td>
<td></td>
</tr>
<tr>
<td>add %l2, %l1, %o1</td>
<td></td>
</tr>
<tr>
<td>sethi %hi(0x2000), %l7</td>
<td></td>
</tr>
<tr>
<td>or %l7, %o2, %l1</td>
<td></td>
</tr>
<tr>
<td>mov %l6, %o4</td>
<td></td>
</tr>
<tr>
<td>mov %o5, %o2</td>
<td></td>
</tr>
<tr>
<td>mov %o6, %o3</td>
<td></td>
</tr>
<tr>
<td>mov 0x5, %o2</td>
<td></td>
</tr>
<tr>
<td>sethi %hi(0x24000000), %o3</td>
<td></td>
</tr>
<tr>
<td>or %o3, %o2, %o4</td>
<td></td>
</tr>
<tr>
<td>mov %l6, %o4</td>
<td></td>
</tr>
</tbody>
</table>

Instruction Scheduling (1)

- Map instructions to time slots on issue units (and resources), such that no hazards occur
  → Global reservation table, resource usage map
- Example without data dependences:

Instruction Scheduling (2)

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

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 of List Scheduling Algorithm

- The level of a task (i.e., instruction) node is the maximal number of nodes that are passed on the way to the final node, itself included.

- The algorithm:
  - The level of each node is used as priority.
  - When a processor/function unit is free, assign the unexecuted task which has highest priority and which is ready to be executed.

Example: Topological Sorting (0)
According to Data Dependencies

- Not yet considered
- Data ready (zero-in-degree set)
- Already scheduled, still live
- Already scheduled, no longer referenced

Example: Topological Sorting (1)
According to Data Dependencies

Example: Topological Sorting (2)
According to Data Dependencies

Example: Topological Sorting (3)
According to Data Dependencies

Example: Topological Sorting (4)
According to Data Dependencies

Data ready (zero-in-degree set)
Already scheduled, still live
Already scheduled, no longer referenced

a

Example: Highest Level First algorithm on a tree structured task graph, 3 processor units

Task Graph

Gantt Chart
**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 sorts...
    - Beware of scheduling anomalies with complex reservation tables!

**Software Pipelining**

\[
\text{for } i := 1 \text{ to } n \\
\begin{align*}
&\text{get values; compute; store;} \\
&\text{end for}
\end{align*}
\]

- Iter 1
  - get values 1
  - compute 1
  - store 1
- Iter 2
  - get values 2
  - compute 2
- Iter 3
  - get values 3
  - compute 3
- Iter ...

In parallel

**Software Pipelining of Loops (1)**

\[
\text{loop: get values; compute; store; end for}
\]

- \(i := 1\) to \(n\)
- Get values, compute, store

**Software Pipelining of Loops (2)**

**Software Pipelining of Loops (3)**

**Modulo Scheduling**

- Assume 4 units, fully pipelined delay=2 for all instructions
- Assume 2 processor cycles

**3. Register Allocation**
Global 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 previous lecture)
- Global register allocation on whole procedure body (with minimal spill code) is NP-complete.
  - Can be modeled as a graph coloring problem [Ershov'62] [Cocke'71].

When do Register Allocation

- Register allocation is normally performed at the end of global optimization, when the final structure of the code and all potential use of registers is known.
- It is performed on abstract machine code where you have access to an unlimited number of registers or some other intermediary form of program.
- The code is divided into sequential blocks (basic blocks) with accompanying control flow graph.

Live Range Example

- x is defined
- Use of x
- Last use of x

Interference Graphs

- The live ranges of two variables interfere if their intersection is not empty.
- Each live range builds a node in the interference graph (or conflict graph)
- If two live ranges interfere, an edge is drawn between the nodes.
- Two adjacent nodes in the graph cannot be assigned the same register.

Register Allocation vs Graph Coloring

- Register allocation can be compared with the classic coloring problem.
  - That is, to find a way of coloring - with a maximum of k colors - the interference graph which does not assign the same color to two adjacent nodes.
  - k = the number of registers.
    - On a RISC-machine there are, for example, 16 or 32 general registers. Certain methods use some registers for other tasks, e.g., for spill code.
  - Determining whether a graph is colorable using k colors is NP-complete for k>3
    - In other words, it is unmanageable always to find an optimal solution.
Register Allocation by Graph Coloring

**Step 1:** Given a program with symbolic registers s1, s2, ...
- Determine live ranges of all variables

```
i = c+4; load @(fp), s1 ! c
add s1, s1, s2
store s3, 4(fp) ! d
```

**Step 2:** Build the Register Interference Graph
- Undirected edge connects two symbolic registers (si, sj) if live ranges of si and sj overlap in time
- Reserved registers (e.g., fp) interfere with all symbolic registers

**Reg. Alloc. by Graph Coloring Cont.**

**Step 3:** Color the register interference graph with k colors, where k = available registers.
- If not possible: pick a victim si to spill, generate spill code (store after def., reload before use)
  This may remove some interferences.
  Rebuild the register interference graph + repeat Step 3...

```
i = c+4; load @(fp), s1 ! c
add s1, s1, s2
store s3, 4(fp) ! d
```

**Coloring a Graph with k Colors**
- NP-complete for k > 3
- Chromatic number χ(G) = minimum number of colors to color a graph G
- χ(G) >= c if the graph contains a c-clique
  - A c-clique is a completely connected subgraph of c nodes

**Chaitin's heuristic (1981):**
```
S ← {s1, s2, ...} // set of spill candidates
while (S not empty)
  choose some s in S.
  if s has less than k neighbors in the graph
    delete s (and incident edges) from the graph
  else
    modify the graph (spill, split, coalesce ... nodes)
    and restart.
  // once we arrive here, the graph is empty:
  color the nodes greedily in reverse order of removal.
```

**Register Allocation for Loops (1)**

- Interference graphs have some weaknesses:
  - Imprecise information on how and when live ranges interfere.
  - No special consideration is taken of loop variables' live ranges (except when calculating priority).
- Instead, in a cyclic interval graph:
  - The time relationships between the live ranges are explicit.
  - Live ranges are represented for a variable whose live range crosses iteration limits by cyclic intervals.

**Chaitin’s Register Allocator (1981):**
- Notation for cyclic live intervals for loops:
  - Intervals for loop variables which do not cross the iteration limit are included precisely once.
  - Intervals which cross the iteration limit are represented as an interval pair, cyclic interval: ([0, t'), [t, t')]

```
This register interference graph cannot be colored with less than 4 colors, as it contains a 4-clique
```
Register Allocation for Loops (2)

Circular edge graph
Only 3 interferences at the same time

Traditional interference graph,
all variables interfere, 4 registers needed

Register Allocation for Loops (3)

Example:
x3 = 7
for i = 1 to 100 {
    x1 = x3 + 2
    x2 = x1 + x3
    x3 = x2 + x1
}
y = x3 + 42

Control flow graph

Live ranges (loop only)
cyclic intervals
e.g. for i: [2, 4], [5, 6]
x1: [2, 4], x2: [3, 5], x3: [0, 3], [4, 6]

At most 3 values live at a time
3 registers sufficient

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

Live Range Coalescing/Combining

(Reduces Register Needs)

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

Phase Ordering Problems

4. Phase Ordering Problems and Integrated Code Generation
Phase Ordering Problems (1)

Instruction scheduling vs. register allocation

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

(b) Register allocation first: Reuse of same register by different values introduces "artificial" data dependences → constrains scheduler

5. Integrated Code Generation

Combination of several IR-sequence optimization problems:

Our Project at PELAB (Kessler): OPTIMIST

Retargetable integrated code generator

Open Source:
www.ida.liu.se/~chrke/optimist