ILP in real code

- Based on all kinds of ideal assumptions. Further limited by:
  - realistic branch prediction
  - finite renaming registers
  - imperfect alias analysis for memory operations
Exposing More ILP

• These techniques were originally motivated by VLIW, which needs tons of ILP to work at all – but useful for superscalar/dynamic/speculative processors, as well.

• Software Techniques
  – Software Pipelining
  – Trace Scheduling

• Hardware/Software Technique
  – Predicated execution
  – Simultaneous Multithreading

Compiler support for ILP: Software Pipelining

• Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations

• Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop

Software Pipelining Example

<table>
<thead>
<tr>
<th>Unrolled 3 times</th>
<th>Software Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 0</td>
<td>LD F0,0(R1)</td>
</tr>
<tr>
<td></td>
<td>ADDD F4,F0,F2</td>
</tr>
<tr>
<td></td>
<td>LD F0,-8(R1)</td>
</tr>
<tr>
<td>Iteration 1</td>
<td>LD F0,0(R1)</td>
</tr>
<tr>
<td></td>
<td>ADDD F4,F0,F2</td>
</tr>
<tr>
<td></td>
<td>LD F0,-8(R1)</td>
</tr>
<tr>
<td>Iteration 2</td>
<td>LD F0,0(R1)</td>
</tr>
<tr>
<td></td>
<td>ADDD F4,F0,F2</td>
</tr>
<tr>
<td></td>
<td>LD F0,-8(R1)</td>
</tr>
<tr>
<td>Iteration 3</td>
<td>LD F0,0(R1)</td>
</tr>
<tr>
<td></td>
<td>ADDD F4,F0,F2</td>
</tr>
<tr>
<td></td>
<td>LD F0,-8(R1)</td>
</tr>
</tbody>
</table>

Compiler Support for ILP: Trace Scheduling

• Creates long basic blocks by finding long paths in the code
Trace Scheduling

• Parallelism across IF branches vs. LOOP branches
• Two steps:
  – Trace Selection
    • Find likely sequence of basic blocks (trace) of (statically predicted) long sequence of straight-line code
  – Trace Compaction
    • Squeeze trace into few VLIW instructions
    • Need bookkeeping code in case prediction is wrong

Predication: HW support for More ILP

• Avoid branch prediction by turning branches into conditionally executed instructions: (aka predicated instructions)
  add c, a, b (x) => if (x) then a = b + c else NOP
  – If false, then neither store result nor cause exception
  – Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr, IA64 can predicate any instruction (even have multiple predicates)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld F2, 0(R2)</td>
<td>load F2 from memory offset 0</td>
</tr>
<tr>
<td>add F4, F2, F0</td>
<td>add F4 to F2 and store result in F0</td>
</tr>
<tr>
<td>muld F6, F4, F4</td>
<td>multiply F6 by F4 and store result in F4</td>
</tr>
<tr>
<td>beqz R3, go_on</td>
<td>branch if (z) to go_on if condition is zero</td>
</tr>
<tr>
<td>addi R2, R2, #8</td>
<td>add 8 to R2</td>
</tr>
<tr>
<td>go_on: addi R2, R2, #8</td>
<td>add 8 to R2</td>
</tr>
<tr>
<td>bnez F6, loop</td>
<td>branch if zero to loop if F6 is zero</td>
</tr>
</tbody>
</table>

Predicated Execution

• Drawbacks to conditional instructions
  – Still takes a clock & alu even if “annulled”
  – Stall if condition evaluated late
  – Complex conditions reduce effectiveness; condition becomes known late in pipeline
  – requires more operands! Typically only available as conditional move.
• Advantages
  – eliminate prediction, misprediction
  – longer basic blocks, ...

• Critical technology for VLIW, sw pipelining. Why?
Pentium Pro (II, III, etc.) microarchitecture

- 40 *uncommitted* instructions
- 20 *unissued* instructions

Pentium 4 microarchitecture

- 126 *in-flight* instructions (ROB size)
Pentium 4 Summary

- 20-stage pipeline
- IA32 (x86) ISA translated to RISC-like uops
- Uops stored in trace cache
- Decode/retire 3 uops/cycle
- Execute 6 uops/cycle
- Dynamically scheduled
- Explicit Register Renaming
- Simultaneous Multithreading (hyper-threading)

ILP Summary

- Parallelism is absolutely critical to modern computer system performance, but at a very fine level.
- Mechanisms that create, or expose parallelism: loop unrolling, software pipelining, code motion
- Mechanisms that allow the machine to exploit ILP: pipelining, superscalar, dynamic scheduling, speculative execution

Simultaneous Multithreading


Motivation

- Modern processors fail to utilize execution resources well.
- There is no single culprit.
- Attacking the problems one at a time (e.g., specific latency-tolerance solutions) always has limited effectiveness.
- However, a general latency-tolerance solution which can hide all sources of latency can have a large impact on performance.
**Simultaneous Multithreading**

- **Issue Slots**
- **Thread 1**
- **Thread 2**
- **Thread 3**
- **Thread 4**
- **Thread 5**

**Goals**

We had three primary goals for this architecture:

1. Minimize the architectural impact on conventional superscalar design.
3. Achieve significant throughput gains with many threads.

**The Potential for SMT**

**A Conventional Superscalar Architecture**
An SMT Architecture

- Fetch up to 8 instructions per cycle
- Out-of-order, speculative execution
- Issue 3 floating point, 6 integer instructions per cycle

Performance of the Naive Design

- Fetch throughput (4.2 instructions per cycle when queue not full)

Bottlenecks of the Baseline Architecture

- Instruction queue full conditions (12-21% of cycles)
  - Lack of parallelism in the queue.
- Fetch throughput (4.2 instructions per cycle when queue not full)

Improving Fetch Throughput

- The fetch unit in an SMT architecture has two distinct advantages over a conventional architecture.
  1. Can fetch from multiple threads at once.
  2. Can choose which threads to fetch.
**Improved Fetch Performance**

- Fetching from 2 threads/cycle achieved most of the performance from multiple-thread fetch.
- Fetching from the thread(s) which have the fewest unissued instructions in-flight significantly increases parallelism and throughput.

**This SMT Architecture, then:**

- Borrows heavily from conventional superscalar design.
- Minimizes the impact on single-thread performance
- Achieves significant throughput gains over the superscalar (2.5X, up to 5.4 IPC).

**Commercial SMT**

- Alpha 21464 (©)
- Intel Pentium 4 “hyper-threading” processor.
- IBM Power 5 – 2 cores, 2 SMT threads/core
- IBM Power 6 – again, 2 cores, 2 SMT threads/core
- IBM Power 7 – 8 cores, 4 SMT threads/core
- Sun Niagara T1 (2006) – 8 cores, 4 threads/core (SMT?)
- Sun Niagara T2 – 8 cores, 8 threads/core
- Intel Nehalem (core i7) 4-8 cores, 2 SMT threads/core