Static Superscalar, VLIW and EPIC

Two Strategies for Higher ILP

Modern superscalar processors: dynamically scheduled, speculative execution, branch prediction, dynamic memory disambiguation, non-blocking cache ⇒ More and more hardware functionalities AND complexities

- Another direction: Let compiler take the complexity
  - Simple hardware, smart compiler
  - Static Superscalar, VLIW, EPIC

MIPS Pipeline with pipelined multi-cycle Operations

Pipelined implementations ex: 7 outstanding MUL, 4 outstanding Add, unpipelined DIZV
In-order execution, out-of-order completion
Tomasulo w/o ROB: out-of-order execution, out-of-order completion, in-order commit

More Hazards Detection and Forwarding

- Assume checking hazards at ID (the simple way)
- Structural hazards
  - Hazards at WB: Track usages of registers at ID, stall instructions if hazards is detected
  - Separate int and fp registers to reduce hazards
- RAW hazards: Check source registers with all EX stages except the last ones.
  - A dependent instruction must wait for the producing instruction to reach the last stage of EX
  - EX: check with ID/A1, A1/A2, A2/A3, but not A4/MEM.
- WAW hazards
  - Instructions reach WB out-of-order
  - check with all multi-cycle stages (A1-A4, D, M1-M7) for the same dest register

Out-of-order completion complicates the maintenance of precise exception
- More forwarding data paths

Complier Optimization

- Example: add a scalar to a vector:
  for (i=1000; i>0; i=i–1)
  x[i] = x[i] + s;

- MIPS code
  Loop: L.D F0,0(R1) ;F0=vector element
  stall for L.D, assume 1 cycles
  ADD.D F4,F0,F2 ;add scalar from F2
  stall for ADD, assume 2 cycles
  S.D 0(R1),F4 ;store result
  DSUBUI R1,R1,8 ;decrement pointer
  BNEZ R1,Loop ;branch R1!=zero
  stall for taken branch, assume 1 cycle

Loop unrolling

1 Loop: L.D F0,0(R1)
2 ADD.D F4,F0,F2 ;drop DSUBUI & BNEZ
3 S.D 0(R1),F4 ;drop DSUBUI & BNEZ
4 L.D F6,=0(R1)
5 ADD.D F9,F6,F2
6 S.D =0(R1),F6 ;drop DSUBUI & BNEZ
7 L.D F10,=0(R1)
8 ADD.D F12,F10,F2
9 S.D =16(R1),F12 ;drop DSUBUI & BNEZ
10 L.D F14,=0(R1)
11 ADD.D F16,F14,F2
12 S.D F14(R1),F16
13 DSUBUI R1,R1, #32 ;alter to 4*8
14 BNEZ R1,LOOP
15 NOP

1 cycle stall
2 cycles stall
**Unrolled Loop That Minimizes Stalls**

1. Loop: L.D F0,0(R1)
2. L.D F6,-8(R1)
3. L.D F10,-16(R1)
4. L.D F14,-24(R1)
5. ADD.D F4,F0,F2
6. ADD.D F8,F6,F2
7. ADD.D F12,F10,F2
8. ADD.D F16,F14,F2
9. S.D 0(R1),F4
10. S.D -8(R1),F8
11. S.D -16(R1),F12
12. DSUBUI R1,R1,#32
13. BNEZ R1,LOOP
14. S.D 0(R1),F16 ; delayed branch slot

Called code movement
- Moving store past DSUBUI
- Moving loads before stores

**Register Renaming**

1. Loop: L.D F0,0(R1)
2. ADD.D F4,F0,F2
3. S.D 0(R1),F4
4. L.D F6,-8(R1)
5. ADD.D F4,F0,F2
6. S.D 0(R1),F4
7. L.D F10,-16(R1)
8. ADD.D F4,F0,F2
9. S.D 0(R1),F4
10. L.D F14,-24(R1)
11. ADD.D F4,F0,F2
12. S.D 0(R1),F4
13. DSUBUI R1,R1,#32
14. BNEZ R1,LOOP
15. NOP

Original register renaming

**VLIW: Very Large Instruction Word**

Static Superscalar: hardware detects hazard, compiler determines scheduling
VLIW: compiler takes both jobs
- Each "instruction" has explicit coding for multiple operations
- There is no or only partial hardware hazard detection
  - No dependence check logic for instruction issued at the same cycle
  - Wide instruction format allows theoretically high ILP
- Tradeoff instruction space for simple decoding
  - But have to fill with NOOP if no enough operations are found

**VLIW Example: Loop Unrolling**

<table>
<thead>
<tr>
<th>Memory reference</th>
<th>Memory reference</th>
<th>FP operation</th>
<th>Int. op/ Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0(R1)</td>
<td>L.D F4,4(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>S.D 0(R1),F4</td>
</tr>
<tr>
<td>L.D F6,-8(R1)</td>
<td>L.D F22,40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F26,48(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F16,F4,F2</td>
</tr>
<tr>
<td>L.D F14,-24(R1)</td>
<td>L.D F10,16(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F24,F2,F2</td>
</tr>
<tr>
<td>S.D 0(R1),F4</td>
<td>S.D 4(R1),F4</td>
<td>ADD.D F26,F2,F2</td>
<td></td>
</tr>
<tr>
<td>S.D -8(R1),F8</td>
<td>ADD.D F26,F2,F2</td>
<td>S.D 0(R1),F4</td>
<td></td>
</tr>
<tr>
<td>S.D -16(R1),F12</td>
<td>ADD.D F26,F2,F2</td>
<td>ADD.D F26,F2,F2</td>
<td></td>
</tr>
<tr>
<td>S.D -24(R1),F16</td>
<td>S.D 40(R1),F24</td>
<td>DSUBUI R1,1448</td>
<td></td>
</tr>
<tr>
<td>S.D -32(R1),F20</td>
<td>S.D 0(R1),F28</td>
<td>DSUBUI R1,1,322</td>
<td></td>
</tr>
<tr>
<td>S.D -40(R1),F24</td>
<td>BNEZ R1,LOOP</td>
<td>S.D 0(R1),F28</td>
<td></td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency
Note: Need more registers in VLIW (15 in this example)

**Scheduling Across Branches**

- Local scheduling or basic block scheduling
  - Typically in a range of 5 to 20 instructions
  - Unrolling may increase basic block size to facilitate scheduling
  - However, what happens if branches exist in loop body?
- Global scheduling: moving instructions across branches (i.e., cross basic blocks)
  - We cannot change data flow with any branch outputs
  - How to guarantee correctness?
  - Increase scheduling scope: trace scheduling, superblock, predicted execution, etc.

**Problems with First Generation VLIW**

- Increase in code size
  - Wasted issue slots are translated to no-ops in instruction encoding
  - About 50% instructions are no-ops
  - The increase is from 5 instructions to 45 instructions!
- Operated in lock-step; no hazard detection HW
  - Any function unit stalls → The entire processor stalls
  - Compiler can schedule around function unit stalls; but how about cache miss?
  - Compilers are not allowed to speculate!
- Binary code compatibility
  - Re-compile if #FU or any FU latency changes; not a major problem today
EPIC/IA-64: Motivation in 1989

"First, it was quite evident from Moore's law that it would soon be possible to fit an entire, highly parallel, ILP processor on a chip. Second, we believed that the ever-increasing complexity of superscalar processors would have a negative impact upon their clock rate, eventually leading to a leveling off of the rate of increase in microprocessor performance."

Schlansker and Rau, Computer Feb. 2000

Obvious today: Think about the complexity of P4, 21264, and other superscalar processor; processor complexity has been discussed in many papers since mid-1990s

Agarwal et al., "Clock rate versus IPC: The end of the road for conventional microarchitectures," ISCA 2000

---

EPIC Main ideas

Compile does the scheduling
  - Permitting the compiler to play the statistics (profiling)

Hardware supports speculation
  - Addressing the branch problem: predicted execution and many other techniques
  - Addressing the memory problem: cache specifiers, prefetching, speculation on memory alias

---

Example: IF-conversion

Use of predicated execution to perform if-conversion: Eliminate the branch and produces just one basic block containing operations guarded by the appropriate predicates. Schlansker and Rau, 2000

---

Memory Issues

- Cache specifiers: compiler indicates cache location in load/store; (use analytical models or profiling to find the answers?)
- Compiler may actively remove data from cache or put data with poor locality into a special cache; reducing cache pollution
- Compiler can speculate that memory alias does not exist thus it can reorder loads and stores
  - Hardware detects any violations
  - Compiler then fixes up

---

Itanium: First Implementation by Intel

- Itanium™ is name of first implementation of IA-64 (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-issue, 10-stage pipeline at 800Mhz on 0.18 µ process
  - 128 64-bit integer registers + 128 82-bit floating point registers
  - Hardware checks dependencies
  - Predicated execution
  - 128 bit Bundle: 5-bit template + 3 46-bit instructions
    - Two bundles can be issued together in Itanium
Comments on Itanium

Remarkably, the Itanium has many of the features more commonly associated with the dynamically-scheduled pipelines.

Performance: 800MHz Itanium, 1GHz 21264, 2GHz P4
- SPEC Int: 85% 21264, 60% P4
- SPEC FP: 108% P4, 120% 21264
- Power consumption: 178% of P4 (watt per FP op)

Surprising that an approach whose goal is to rely on compiler technology and simpler HW seems to be at least as complex as dynamically scheduled processors!