ECE/CS 552: Pipelining to Superscalar

Instructor: Mikko H Lipasti

Fall 2010
University of Wisconsin-Madison

Lecture notes based on notes by John P. Shen
Updated by Mikko Lipasti

Pipelining to Superscalar

- Forecast
  - Real pipelines
  - IBM RISC Experience
  - The case for superscalar
  - Instruction-level parallel machines
  - Superscalar pipeline organization
  - Superscalar pipeline design

MIPS R2000/R3000 Pipeline

<table>
<thead>
<tr>
<th>Stage</th>
<th>Phase</th>
<th>Function performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>$\phi_1$</td>
<td>Translate virtual instr. addr. using TLB</td>
</tr>
<tr>
<td></td>
<td>$\phi_2$</td>
<td>Access L-cache</td>
</tr>
<tr>
<td>RD</td>
<td>$\phi_1$</td>
<td>Return instruction from I-cache, check tags &amp; parity</td>
</tr>
<tr>
<td></td>
<td>$\phi_2$</td>
<td>Read RF; if branch, generate target</td>
</tr>
<tr>
<td>ALU</td>
<td>$\phi_1$</td>
<td>Start ALU op; if branch, check condition</td>
</tr>
<tr>
<td></td>
<td>$\phi_2$</td>
<td>Finish ALU op; if load, translate addr</td>
</tr>
<tr>
<td>MEM</td>
<td>$\phi_1$</td>
<td>Access D-cache</td>
</tr>
<tr>
<td></td>
<td>$\phi_2$</td>
<td>Return data from D-cache, check tags &amp; parity</td>
</tr>
<tr>
<td>WB</td>
<td>$\phi_1$</td>
<td>Write RF</td>
</tr>
<tr>
<td></td>
<td>$\phi_2$</td>
<td></td>
</tr>
</tbody>
</table>

Intel i486 5-stage Pipeline

<table>
<thead>
<tr>
<th>Stage</th>
<th>Function Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Fetch instruction from 32B prefetch buffer (separate fetch unit fills and flushes prefetch buffer)</td>
</tr>
<tr>
<td>ID-1</td>
<td>Translate instr. into control signals or microcode address, initiate address generation and memory access</td>
</tr>
<tr>
<td>ID-2</td>
<td>Access microcode memory, send microinstruction(s) to execute unit</td>
</tr>
<tr>
<td>EX</td>
<td>Execute ALU and memory operations</td>
</tr>
<tr>
<td>WB</td>
<td>Write back to RF</td>
</tr>
</tbody>
</table>

IBM RISC Experience [Agerwala and Cocke 1987]

- Internal IBM study: Limits of a scalar pipeline?
- Memory Bandwidth
  - Fetch 1 instr/cycle from I-cache
  - 40% of instructions are load/store (D-cache)
- Code characteristics (dynamic)
  - Loads ~ 25%
  - Stores 15%
  - ALU/RR ~ 40%
  - Branches & jumps ~ 20%
    - 1/3 unconditional (always taken)
    - 1/3 conditional taken, 1/3 conditional not taken

IBM Experience

- Cache Performance
  - Assume 100% hit ratio (upper bound)
  - Cache latency: 1$\rightarrow$D$\rightarrow$1 cycle default
- Load and branch scheduling
  - Loads
    - 25% cannot be scheduled (delay slot empty)
    - 65% can be moved back 1 or 2 instructions
    - 10% can be moved back 1 instruction
  - Branches & jumps
    - Unconditional – 100% schedulable (fill one delay slot)
    - Conditional – 50% schedulable (fill one delay slot)
CPI Optimizations

- Goal and impediments
  - CPI = 1, prevented by pipeline stalls
  - No cache bypass of RF, no load/branch scheduling
    - Load penalty: 2 cycles: 0.25 x 2 = 0.5 CPI
    - Branch penalty: 2 cycles: 0.2 x 2/3 x 2 = 0.27 CPI
    - Total CPI: 1 + 0.5 + 0.27 = 1.77 CPI
- Bypass, no load/branch scheduling
  - Load penalty: 1 cycle: 0.25 x 1 = 0.25 CPI
  - Total CPI: 1 + 0.25 + 0.27 = 1.52 CPI

More CPI Optimizations

- Bypass, scheduling of loads/branches
  - Load penalty:
    - 65% + 10% = 75% moved back, no penalty
    - 25% => 1 cycle penalty
    - 0.25 x 0.25 x 1 = 0.0625 CPI
  - Branch penalty:
    - 1/3 unconditional 100% schedulable => 1 cycle
    - 1/3 cond. not-taken, => no penalty (predict not-taken)
    - 1/3 cond. Taken, 50% schedulable => 1 cycle
    - 1/3 cond. Taken, 50% unschedulable => 2 cycles
    - 0.20 x (1/3 x 1 + 1/3 x 0.5 x 1 + 1/3 x 0.5 x 2) = 0.167
  - Total CPI: 1 + 0.063 + 0.167 = 1.23 CPI

Simplify Branches

- Assume 90% can be PC-relative
  - No register indirect, no register access
  - Separate adder (like MIPS R3000)
  - Branch penalty reduced
- Total CPI: 1 + 0.063 + 0.085 = 1.15 CPI = 0.87 IPC

<table>
<thead>
<tr>
<th>PC-relative</th>
<th>Schedulable</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes (90%)</td>
<td>Yes (50%)</td>
<td>0 cycle</td>
</tr>
<tr>
<td>Yes (90%)</td>
<td>No (50%)</td>
<td>1 cycle</td>
</tr>
<tr>
<td>No (10%)</td>
<td>Yes (50%)</td>
<td>1 cycle</td>
</tr>
<tr>
<td>No (10%)</td>
<td>No (50%)</td>
<td>2 cycles</td>
</tr>
</tbody>
</table>

Processor Performance

\[
\text{Processor Performance} = \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- In the 1980’s (decade of pipelining):
  - CPI: 5.0 ⇒ 1.15
- In the 1990’s (decade of superscalar):
  - CPI: 1.15 ⇒ 0.5 (best case)

Revisit Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - f + \frac{f}{v}}
\]

\[
\lim_{v \to \infty} \frac{1}{1 - f + \frac{f}{v}} = \frac{1}{1 - f}
\]

Revisit Amdahl’s Law

- Sequential bottleneck
- Even if v is infinite
  - Performance limited by nonvectorizable portion (1-f)
Pipelined Performance Model

- g = fraction of time pipeline is filled
- 1-g = fraction of time pipeline is not filled (stalled)

Motivation for Superscalar

- Tyranny of Amdahl’s Law [Bob Colwell]
  - When g is ever slightly below 100%, a big performance hit will result
  - Stalled cycles are the key adversary and must be minimized as much as possible

Superscalar Proposal

- Moderate tyranny of Amdahl’s Law
  - Ease sequential bottleneck
  - More generally applicable
  - Robust (less sensitive to f)
  - Revised Amdahl’s Law:

\[ \text{Speedup} = \frac{1}{(1 - f) + \frac{f}{s}} \]

Limits on Instruction Level Parallelism (ILP)

<table>
<thead>
<tr>
<th>Authors</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hawe and Kraft (1994)</td>
<td>1.59</td>
</tr>
<tr>
<td>Bob and Vaipera (1992)</td>
<td>1.41</td>
</tr>
<tr>
<td>Coghill and Harvey (1991)</td>
<td>1.44</td>
</tr>
<tr>
<td>Tjaden and Flynn (1975)</td>
<td>1.56</td>
</tr>
<tr>
<td>Lee (1984)</td>
<td>2.00</td>
</tr>
<tr>
<td>Soh and Vajapeyam (1987)</td>
<td>2.00</td>
</tr>
<tr>
<td>Tjaden and Flynn (1970)</td>
<td>2.15</td>
</tr>
<tr>
<td>Kop. (1991)</td>
<td>2.50</td>
</tr>
<tr>
<td>Acosta et al. (1986)</td>
<td>2.79</td>
</tr>
<tr>
<td>Wedig (1982)</td>
<td>3.00</td>
</tr>
<tr>
<td>Butler et al. (1991)</td>
<td>5.80</td>
</tr>
<tr>
<td>Melvin and Patt (1991)</td>
<td>6.00</td>
</tr>
<tr>
<td>Smith et al. (1991)</td>
<td>7.00</td>
</tr>
<tr>
<td>Nicolau and Fisher (1987)</td>
<td>90 (Fisher’s optimism)</td>
</tr>
<tr>
<td>Riseman and Foster (1972)</td>
<td>51 (no control dependences)</td>
</tr>
<tr>
<td>Kuck et al. (1972)</td>
<td>8.00</td>
</tr>
</tbody>
</table>
Superscalar Proposal
- Go beyond single instruction pipeline, achieve IPC > 1
- Dispatch multiple instructions per cycle
- Provide more generally applicable form of concurrency (not just vectors)
- Geared for sequential code that is hard to parallelize otherwise
- Exploit fine-grained or instruction-level parallelism (ILP)

Classifying ILP Machines
[Jouppi, DECWRL 1991]
- Baseline scalar RISC
  - Issue parallelism = IP = 1
  - Operation latency = OP = 1
  - Peak IPC = 1

- Superpipelining: cycle time = 1/m of baseline
  - Issue parallelism = IP = 1 inst / minor cycle
  - Operation latency = OP = m minor cycles
  - Peak IPC = m instr / major cycle (m x speedup?)

- Superscalar: Issue parallelism = IP = n inst / cycle
  - Operation latency = OP = 1 cycle
  - Peak IPC = n instr / cycle (n x speedup?)

- VLIW: Very Long Instruction Word
  - Issue parallelism = IP = n inst / cycle
  - Operation latency = OP = 1 cycle
  - Peak IPC = n instr / cycle = 1 VLIW / cycle
Superscalar vs. Superpipelined

- Roughly equivalent performance
  - If $n = m$ then both have about the same IPC
  - Parallelism exposed in space vs. time

### Time in Cycles (of Base Machine)
```
0  1  2  3  4  5  6  7  8  9
```

#### Key:
- IFetch
- Decode
- Execute
- Writeback
- Commit
- Memory

Superscalar Challenges

- I-cache
- Fetch
- Decode
- Branch Predictor
- Instruction Buffer
- Commit
- D-cache
- Store Queue
- Reorder Buffer
- Integer Floating-point Media Memory Register Data Flow

Memory Stalls Flow