CS3014: Computer Architecture

Superscalar Pipelines

Slides developed by Joe Devietti, Milo Martin & Amir Roth at U. Penn with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood
An Opportunity...

• But consider:
  
  ADD r1, r2 -> r3  
  ADD r4, r5 -> r6  
  • Why not execute them \textit{at the same time}? (We can!)

• What about:
  
  ADD r1, r2 -> r3  
  ADD r4, r3 -> r6  
  • In this case, \textit{dependences} prevent parallel execution

• What about three instructions at a time?  
  • Or four instructions at a time?
What Checking Is Required?

- For two instructions: 2 checks
  
  ADD src\textsubscript{1\textsubscript{1}}, src\textsubscript{2\textsubscript{1}} \rightarrow dest\textsubscript{1\textsubscript{1}}
  
  ADD src\textsubscript{1\textsubscript{2}}, src\textsubscript{2\textsubscript{2}} \rightarrow dest\textsubscript{2\textsubscript{2}} (2 checks)

- For three instructions: 6 checks
  
  ADD src\textsubscript{1\textsubscript{1}}, src\textsubscript{2\textsubscript{1}} \rightarrow dest\textsubscript{1\textsubscript{1}}
  
  ADD src\textsubscript{1\textsubscript{2}}, src\textsubscript{2\textsubscript{2}} \rightarrow dest\textsubscript{2\textsubscript{2}} (2 checks)
  
  ADD src\textsubscript{1\textsubscript{3}}, src\textsubscript{2\textsubscript{3}} \rightarrow dest\textsubscript{3\textsubscript{3}} (4 checks)

- For four instructions: 12 checks
  
  ADD src\textsubscript{1\textsubscript{1}}, src\textsubscript{2\textsubscript{1}} \rightarrow dest\textsubscript{1\textsubscript{1}}
  
  ADD src\textsubscript{1\textsubscript{2}}, src\textsubscript{2\textsubscript{2}} \rightarrow dest\textsubscript{2\textsubscript{2}} (2 checks)
  
  ADD src\textsubscript{1\textsubscript{3}}, src\textsubscript{2\textsubscript{3}} \rightarrow dest\textsubscript{3\textsubscript{3}} (4 checks)
  
  ADD src\textsubscript{1\textsubscript{4}}, src\textsubscript{2\textsubscript{4}} \rightarrow dest\textsubscript{4\textsubscript{4}} (6 checks)

- Plus checking for load-to-use stalls from prior \( n \) loads
What Checking Is Required?

• For two instructions: 2 checks
  \[
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_1 \\
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_2 \quad (2 \text{ checks})
  \]

• For three instructions: 6 checks
  \[
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_1 \\
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_2 \quad (2 \text{ checks}) \\
  \text{ADD } \text{src}_3, \text{src}_2 \rightarrow \text{dest}_3 \quad (4 \text{ checks})
  \]

• For four instructions: 12 checks
  \[
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_1 \\
  \text{ADD } \text{src}_1, \text{src}_2 \rightarrow \text{dest}_2 \quad (2 \text{ checks}) \\
  \text{ADD } \text{src}_3, \text{src}_2 \rightarrow \text{dest}_3 \quad (4 \text{ checks}) \\
  \text{ADD } \text{src}_4, \text{src}_2 \rightarrow \text{dest}_4 \quad (6 \text{ checks})
  \]

• Plus checking for load-to-use stalls from prior \(n\) loads
A Typical Dual-Issue Pipeline

- Fetch an entire 16B or 32B cache block
  - 4 to 8 instructions (assuming 4-byte average instruction length)
  - Predict a single branch per cycle
- Parallel decode
  - Need to check for conflicting instructions
    - **Is output register of I₁ is an input register to I₂?**
  - Other stalls, too (for example, load-use delay)
A Typical Dual-Issue Pipeline

- Multi-ported register file
  - Larger area, latency, power, cost, complexity
- Multiple execution units
  - Simple adders are easy, but bypass paths are expensive
- Memory unit
  - Single load per cycle (stall at decode) probably okay for dual issue
  - Alternative: add a read port to data cache
    - Larger area, latency, power, cost, complexity
Superscalar Implementation Challenges
Superscalar Challenges

• **Superscalar instruction fetch**
  • Modest: fetch multiple instructions per cycle
  • Aggressive: buffer instructions and/or predict multiple branches

• **Superscalar instruction decode**
  • Replicate decoders

• **Superscalar instruction issue**
  • Determine when instructions can proceed in parallel
  • More complex stall logic - $O(N^2)$ for $N$-wide machine
  • Not all combinations of types of instructions possible

• **Superscalar register read**
  • Port for each register read (4-wide superscalar $\Rightarrow$ 8 read “ports”)
  • Each port needs its own set of address and data wires
    • Latency & area $\propto$ #ports$^2$
Superscalar Challenges

• **Superscalar instruction execution**
  • Replicate arithmetic units (but not all, say, integer divider)
  • Perhaps multiple cache ports (slower access, higher energy)
    • Only for 4-wide or larger (why? only ~25% are load/store insn)

• **Superscalar register bypass paths**
  • More possible sources for data values
  • \(O(N^2)\) for \(N\)-wide machine

• **Superscalar instruction register writeback**
  • One write port per instruction that writes a register
  • Example, 4-wide superscalar \(\downarrow\) 4 write ports

• **Fundamental challenge:**
  • Amount of ILP (instruction-level parallelism) in the program
Superscalar Register Bypass

• **Flow of data between instructions**
  - Consider the code
    
    \[
    r1 = r3 \times r4; \\
    r7 = r1 + r2;
    \]
  - The second instruction consumes a value computed by the first

• **Simple solution**
  - First instruction writes its result to \( r1 \)
  - Second instruction reads value from \( r1 \)
  - But the write and read take time
  - The write-back pipeline stage normally happens at least one cycle later than the execute
  - Register read normally happens at least one cycle earlier than execute
  - Potential for delay of one or more cycles
Superscalar Register Bypass

- **Flow of data between instructions**
  - Consider the code
    \[
    r1 = r3 * r4; \\
    r7 = r1 + r2; 
    \]
  - The second instruction consumes a value computed by the first

- **Register Bypassing**
  - Hardware mechanism to allow data to flow directly from the output of one instruction to the input of another
  - The result of the first instruction is written to register \( r1 \)
  - But at the same time a second copy of the result is piped directly to the arithmetic unit that consumes the value
  - Requires a hardware interconnection network between the outputs of functional units (such as adders, multipliers) and the inputs of other functional units
Superscalar Register Bypass

- **N² bypass network**
  - (N+1)-input muxes at each ALU input
  - N² point-to-point connections
  - Routing lengthens wires
  - Heavy capacitive load

- And this is just one bypass stage!
  - Even more for deeper pipelines

- One of the big problems of superscalar
  - Why? On the critical path of single-cycle “bypass & execute” loop
Mitigating $N^2$ Bypass & Register File

- **Clustering**: mitigates $N^2$ bypass
  - Group ALUs into $K$ clusters
  - Full bypassing within a cluster
  - Limited bypassing between clusters
    - *With 1 or 2 cycle delay*
      - Can hurt IPC, but faster clock
  - $(N/K) + 1$ inputs at each mux
  - $(N/K)^2$ bypass paths in each cluster

- **Steering**: key to performance
  - Steer dependent insns to same cluster

- **Cluster register file**, too
  - Replicate a register file per cluster
  - All register writes update all replicas
  - Fewer read ports; only for cluster
Another Challenge: Superscalar Fetch

- What is involved in fetching multiple instructions per cycle?
- In same cache block? no problem
  - 64-byte cache block is 16 instructions (~4 bytes per instruction)
  - Favors larger block size (independent of hit rate)
- What if next instruction is last instruction in a block?
  - Fetch only one instruction that cycle
  - Or, some processors may allow fetching from 2 consecutive blocks
- What about taken branches?
  - How many instructions can be fetched on average?
  - Average number of instructions per taken branch?
    - Assume: 20% branches, 50% taken ~10 instructions
- Consider a 5-instruction loop with a 4-issue processor
  - Without smarter fetch, ILP is limited to 2.5 (not 4, which is bad)
Multiple-Issue Implementations

- **Statically-scheduled (in-order) superscalar**
  - What we’ve talked about thus far
    + Executes unmodified sequential programs
      - Hardware must figure out what can be done in parallel
    - E.g., Pentium (2-wide), UltraSPARC (4-wide), Alpha 21164 (4-wide)

- **Very Long Instruction Word (VLIW)**
  - Compiler identifies independent instructions, new ISA
    + Hardware can be simple and perhaps lower power
    - E.g., TransMeta Crusoe (4-wide)

- **Dynamically-scheduled superscalar**
  - Hardware extracts more ILP by on-the-fly reordering
  - Core 2, Core i7 (4-wide), Alpha 21264 (4-wide)
Trends in Single-Processor Multiple Issue

<table>
<thead>
<tr>
<th>Year</th>
<th>486</th>
<th>Pentium I</th>
<th>Pentium II</th>
<th>Pentium 4</th>
<th>Itanium</th>
<th>Itanium II</th>
<th>Core2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1989</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>1993</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2006</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Issue width has saturated at 4-6 for high-performance cores
  - Canceled Alpha 21464 was 8-way issue
  - Not enough ILP to justify going to wider issue
  - Hardware or compiler *scheduling* needed to exploit 4-6 effectively

- For high-performance *per watt* cores (say, smart phones)
  - Typically 2-wide superscalar (but increasing each generation)