Multiple Issue (Superscalar)

- basic pipeline: single, in-order issue
- first extension: multiple issue (superscalar)
  - still in-order
- future topics
  - out-of-order execution
  - fancy static scheduling (i.e., compiler help)

Flynn Meets Fisher

- “Flynn bottleneck”
  - single issue performance limit is CPI = IPC = 1
  - hazards + overhead \( \Rightarrow \) CPI >= 1 (IPC <= 1)
  - diminishing returns from superpipelining
- solution: issue multiple instructions per cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst0</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst2</td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>inst3</td>
<td></td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
</tr>
</tbody>
</table>

- “instruction-level parallelism” [Fisher’81]
- 1st superscalar: IBM America \( \rightarrow \) RS/6000 \( \rightarrow \) POWER1

Instruction Level Parallelism (ILP)

ILP is a property of the software (not the hardware)
- how much parallelism exists among instructions?
  - very dependent on the software
- many possible ways to exploit ILP
  - pipelining
  - superscalar
  - out-of-order execution (dynamic scheduling)
  - compiler scheduling of code (static scheduling)

Readings

H+P
- chapter 3 (not required, but suggested)

------------- future readings below -------------

Readings in Computer Architecture
- Sohi and Vajapeyam: “Instruction Issue Logic”
- Yeager: “The MIPS R10000 Superscalar Microprocessor”

Recent Research Papers
- Complexity-Effective Superscalar (read this one first)
- DIVA
Base Implementation

- statically scheduled (in-order) superscalar
  - executes unmodified sequential programs
  - figures out on its own what can be done in parallel
  - e.g., Sun UltraSPARC, Alpha 21164
  - we’ll start with this one
- rest of chapter 3 will look at dynamic scheduling
- chapter 4 will look at fancier static scheduling

5-Stage Dual-Issue Pipeline

- what is involved in fetching two instructions per cycle?
- decoding two instructions per cycle?
- executing two ALU operations per cycle?
- accessing the data cache twice per cycle?
- writing back two results per cycle?
- what about 4 or 8 instructions per cycle?

Wide Fetch

- if instructions are sequential...
  - and on same cache line ⇒ nothing really
  - and on different cache lines ⇒ banked I$ + combining network
- if instructions are not sequential...
  - more difficult
  - two serial I$ accesses (access1⇒predict target⇒access2)? no
- note: embedded branches OK as long as predicted NT
  - serial access + prediction in parallel
  - if prediction is T, discard serial part after branch

A Solution to Wide Fetch: Trace Cache

- problem: low fetch utilization on taken branches
  - only fetch up to taken branch, remaining fetch slots lost
- trace cache: combine branch predictor with I$
  - [Weiser+Peleg'95, Rotenberg,Bennett+Smith'96]
  - stores dynamic instruction sequences
    - tag: initial PC + directions of embedded branches
    - fetch from trace, but make sure that branch directions were ok
      - typically backed by I$ (in case of trace cache miss)
  - used in Pentium4
    - actually a decoded (µop) trace cache
Trace Cache Example

- instruction cache with 2 instrs per block

<table>
<thead>
<tr>
<th>IS</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>data</td>
<td>inst0 (beq r1, inst4)</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>PC0</td>
<td>inst0,inst1</td>
<td>inst4</td>
<td>F*</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
</tr>
<tr>
<td>PC2</td>
<td>inst2,inst3</td>
<td>inst5</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>PC4</td>
<td>inst4,inst5</td>
<td>inst6</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

- trace-cache with 2 instrs per block

<table>
<thead>
<tr>
<th>T$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>data</td>
<td>inst0 (beq r1, inst4)</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>PC0:1</td>
<td>inst0,inst4</td>
<td>inst4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>PC2:--</td>
<td>inst2,inst3</td>
<td>inst5</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
<tr>
<td>PC5:--</td>
<td>inst5,inst6</td>
<td>inst6</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
</tr>
</tbody>
</table>

Wide Decode

- actually decoding instructions?
  - easy if fixed length instructions (multiple decoders)
  - harder (but possible) if variable length

- reading input register values?
  - $2N$ register read ports (register file read latency ~$2N$)
  - actually less than $2N$, since most values come from bypasses

- what about the stall logic to enforce RAW dependences?

N$^2$ Dependence Cross-Check

- remember stall logic for single issue pipeline
  - $rs1(D) == rd(D/X) \lor rs1(D) == rd(X/M) \lor rs1(D) == rd(M/W)$
  - same for $rs2(D)$
  - full-bypassing reduces to $rs1(D) == rd(D/X) \& \& op(D/X) == LOAD$

- doubling issue width ($N$) quadruples stall logic!
  - not only 2 instructions in $D$, but two instructions in every stage
  - $rs1(D_1) == rd(D/X_1) \& \& op(D/X_1) == LOAD$
  - $rs1(D_1) == rd(D/X_2) \& \& op(D/X_2) == LOAD$
  - repeat for $rs1(D_2)$, $rs2(D_1)$, $rs2(D_2)$
  - also check dependence of 2nd instruction on 1st: $rs1(D_2) == rd(D_1)$

“N$^2$ dependence cross-check”
- for N-wide pipeline, stall (and bypass) circuits grow as $N^2$

Superscalar Stalls

- invariant: stalls propagate upstream to younger instructions

- what if older instruction in issue “pair” (inst0) stalls?
  - younger instruction (inst1) stalls too, cannot pass it

- what if younger instruction (inst1) stalls?
  - can older instruction from next group (inst2) move up?

rigid pipe $\Rightarrow$ no

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst0</td>
<td>F</td>
<td>D</td>
<td>X</td>
</tr>
<tr>
<td>inst1</td>
<td>F</td>
<td>D</td>
<td>d*</td>
</tr>
<tr>
<td>inst2</td>
<td>F</td>
<td>p*</td>
<td>D</td>
</tr>
<tr>
<td>inst3</td>
<td>F</td>
<td>p*</td>
<td>D</td>
</tr>
</tbody>
</table>

fluid pipe $\Rightarrow$ yes

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst0</td>
<td>F</td>
<td>D</td>
<td>X</td>
</tr>
<tr>
<td>inst1</td>
<td>F</td>
<td>D</td>
<td>d*</td>
</tr>
<tr>
<td>inst2</td>
<td>F</td>
<td>D</td>
<td>X</td>
</tr>
<tr>
<td>inst3</td>
<td>F</td>
<td>p*</td>
<td>D</td>
</tr>
</tbody>
</table>
Wide Execute

what is involved in fetching \( N \) instructions per cycle?

- multiple execution units... \( N \) of every kind?
  - \( N \) ALUs? OK, ALUs are small
  - \( N \) FP dividers? no, FP dividers are huge (and fdiv is uncommon)
- typically have some mix (proportional to instruction mix)
  - RS/6000: 1 ALU/memory/branch + 1 FP
  - Pentium: 1 any + 1 ALU (Pentium)
  - Pentium II: 1 ALU/FP + 1 ALU + 1 load + 1 store + 1 branch
  - Alpha 21164: 1 ALU/FP/branch + 2 ALU + 1 load/store

\[ N^2 \text{ Bypass} \]

- \( N^2 \) bypass logic... OK
  - only 5-bit quantities
  - compare to generate 1-bit outcomes
  - similar to stall logic
- \( N^2 \) bypass buses... not even close to OK
  - 32-bit or 64-bit quantities
  - broadcast, route, and multiplex (mux)
  - difficult to lay out and route all the wires
  - wide (SLOW) muxes
  - big design problem today

Alleviating Bypassing with Clustering

- group functional units into clusters
  - full bypass within cluster
  - no bypass between clusters
  - \( \sim (N/k) \) inputs at each mux
  - \( \sim (N/k)^2 \) routed buses in each cluster
- steer instructions to different clusters
  - dependent instructions to same cluster
  - exploit intra-cluster bypass
  - static or dynamic steering is possible
- e.g. Alpha 21164
  - 4-wide, 300MHz
  - full bypass didn’t fit into 1 clock cycle
  - 2 clusters with full intra-cluster bypass

Wide Memory Access

what is involved in accessing memory multiple instructions per cycle?

- multi-banked \( D\)
  - requires bank assignment and conflict-detection logic
- (rough) instruction mix: 20% loads, 15% stores
  - for width \( N \), we need about 0.2*N load ports, 0.15*N store ports
Wide Writeback

• nothing too special, just another port on the register file
  • everything else is taken care of earlier in pipeline

Multiple Issue Summary

• superscalar problem spots
  • fetch, branch prediction ⇒ trace cache?
  • decode ($N^2$ dependence cross-check)
  • execute ($N^2$ bypass) ⇒ clustering?

next up: dynamic scheduling (out-of-order issue)