Instruction Fetch Unit Using I-cache

- Instruction Fetch Unit
- I-cache
- I-TLB
- Decoder
- Register renaming
- Execution units
- Branch Pred
Instruction Fetch Unit Using Trace Cache
Two Traces can have the same tag

- Assume traces ≤ 16 instructions
- Traces B1B2B4 and B1B3B4 have same tag (address L1)
- Differentiated by trace predictor or something else (e.g., # of branches taken)

```
B1: 6 instructions
B2: 5 instructions
B3: 4 instructions
B4: 5 instructions
```

L1: then
    B1: 6 instructions

else
    B2: 5 instructions
    then
    B3: 4 instructions
    else
    B4: 5 instructions
Back-end Operations (OOO)

• Instruction scheduling
  – Detecting a ready instruction: wake-up
  – Maybe more than m ready instructions in an m-way superscalar: need to select

• An often “important“ instruction” is load
  – Load dependencies are bottlenecks
  – Load latencies are variable
  – Does a given load conflict with previous store? Load speculation

• Other optimizations
  – Value prediction??? Critical instructions??? Clustering of functional units???
Reservation Stations and Functional Units

<table>
<thead>
<tr>
<th>Processor</th>
<th>RS type</th>
<th>Number of res. stations</th>
<th>Functional units</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Power PC 620</td>
<td>distributed</td>
<td>15</td>
<td>4(1) 1 1</td>
<td></td>
</tr>
<tr>
<td>IBM Power 4</td>
<td>distributed</td>
<td>31</td>
<td>4(2) 2 2</td>
<td></td>
</tr>
<tr>
<td>Intel P6 (Pentium III)</td>
<td>centralized</td>
<td>20</td>
<td>3(3) 2 4(4)</td>
<td></td>
</tr>
<tr>
<td>Intel Pentium 4</td>
<td>hybrid (1 for mem.op, 1 for rest)</td>
<td>126 (72,54)</td>
<td>5(5) 2 2(6)</td>
<td></td>
</tr>
<tr>
<td>AMD K6</td>
<td>centralized</td>
<td>72</td>
<td>3 3 3</td>
<td></td>
</tr>
<tr>
<td>AMD Opteron</td>
<td>distributed</td>
<td>60</td>
<td>3 3 3</td>
<td></td>
</tr>
<tr>
<td>MIPS 10000</td>
<td>hybrid (1 for int, 1 for mem, 1 for fp)</td>
<td>48 (16,16,16)</td>
<td>2 1 2</td>
<td></td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>hybrid (1 for int /mem 1 for fp)</td>
<td>35 (20,15)</td>
<td>2 2 2</td>
<td></td>
</tr>
<tr>
<td>Ultra SparcIII</td>
<td>In-order queue</td>
<td>20</td>
<td>2 1 3(7)</td>
<td></td>
</tr>
</tbody>
</table>
Wake-up

- If $f$ functional units, then up to $f$ results per cycle
- Hence $f$ comparators per operand in reservation station
- If $w$ reservation stations then need of $2fw$ comparators
  - From previous slide over a 1000 comparators!
- Can be reduced by
  - Res. Stations distributed by function
  - There might not be $f$ broadcast buses
Select

- Hardwired priority
  - Enforced by a hardware encoder: woken-up instruction sends a request for issue to encoder
  - In general “oldest woken-up instruction” first

- Examples of difficulty:
  - Result register name of an instruction must be broadcast one cycle before the result is computed so that a dependent instruction can be woken-up in time to get the forwarded result
  - In case of a cache access, this is speculative so need to be able to recover, i.e., not execute a selected instruction at a given time but let it remain in the instruction window
Load Speculation

- Load = Address computation + Get memory contents
- Two flavors of speculation
  - Address speculation: Used for prefetching (see cache techniques later on)
  - Memory dependence prediction: dependence between loads and previous stores. The so-called memory disambiguation in Intel Core architecture, for example
Store Buffer

• Once the address to where to store has been generated, the store will be put in a store buffer if either
  – The result of the store depends on an uncompleted instruction
  – The result of the store is known but the store instruction is not committed

• An entry in the store buffer consists of:
  – A bit to indicate that the entry is free (state AV)
  – The store has been woken-up, the store address has been computed but the result is not there (state AD)
  – Address and result are there but the store has not been committed (state RE)
  – The store instruction has been committed (state CO)
Load/Store Unit
Load Issue

• Simple scheme: Load and store issue (to AGU) in program order
  – Simplest: Load can issue only if store buffer empty
  – Simpler: load bypassing – load issue if no address conflict with addresses in store buffer
    • Requires to check if preceding store instruction has entered the address in the store buffer
    • If there is a match in state AD or RE the load is aborted (contents discarded)
  – Next: load forwarding
    • Take advantage of states RE and CO and forward result to result register of load.
More load speculation

• Stores issue in program order but a load can issue before some store (i.e., load/store res. station is not a queue)
• Pessimistic approach (previous slide) + check that there is no store left “unissued” in reservation station before the load
  – Used in Pentium
• Optimistic approach: always issue loads
  – Need of a load buffer so we can recover
• Dependence prediction
  – Like optimistic but use of a predictor of memory dependencies and hence fewer recoveries
Example

Prior to this instruction all stores have been committed

\[ \begin{align*}
  i1: \text{st} & \quad R1, \text{memadd1} \\
  i2: \text{st} & \quad R2, \text{memadd2} \\
  i3: \text{ld} & \quad R3, \text{memadd3} \\
  i4: \text{ld} & \quad R4, \text{memadd4}
\end{align*} \]
Example (c’ed)

- **Pessimistic:**
  - no load can issue until i2 has computed its address and put it in store buffer
  - Then i4 can issue
  - i3 will have to wait till i2 has computed result and can forward (state RE)

- **Optimistic**
  - i3 and i4 issue and are put in load buffer.
  - When i1 computes its address, nothing happens in the load buffer
  - When i2 reaches state RE (or AD depending on implementation), i3 and i4 are removed from the load buffer and will have to reissue (i4 because it might depend on i3, again depending on implementation)

- **Dependence prediction**
  - If dependence between i2 and i3 is predicted, i3 cannot issue but i4 can (if not dependent on i3)