Lect. 3: Superscalar Processors

- Pipelining: several instructions are simultaneously at different stages of their execution
- Superscalar: several instructions are simultaneously at the same stages of their execution
- Out-of-order execution: instructions can be executed in an order different from that specified in the program
- Dependences between instructions:
  - Data Dependence (a.k.a. Read after Write - RAW)
  - Control dependence
- Speculative execution: tentative execution despite dependences
A 5-stage Pipeline

IF = instruction fetch (includes PC increment)
ID = instruction decode + fetching values from general purpose registers
EXE = arithmetic/logic operations or address computation
MEM = memory access or branch completion
WB = write back results to general purpose registers
A Pipelining Diagram

- Start one instruction per clock cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>I1</td>
</tr>
<tr>
<td>ID</td>
<td>I1</td>
</tr>
<tr>
<td>EXE</td>
<td>I1</td>
</tr>
<tr>
<td>MEM</td>
<td>I1</td>
</tr>
<tr>
<td>WB</td>
<td>I1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

∴ each instruction still takes 5 cycles, but instructions now complete every cycle: CPI $\rightarrow 1$
Multiple-issue Superscalar

- Start two instructions per clock cycle

<table>
<thead>
<tr>
<th>Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>I1</td>
<td>I3</td>
<td>I5</td>
<td>I7</td>
<td>I9</td>
<td>I11</td>
</tr>
<tr>
<td></td>
<td>I2</td>
<td>I4</td>
<td>I6</td>
<td>I8</td>
<td>I10</td>
<td>I12</td>
</tr>
<tr>
<td>ID</td>
<td>I1</td>
<td>I3</td>
<td>I5</td>
<td>I7</td>
<td>I9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I2</td>
<td>I4</td>
<td>I6</td>
<td>I8</td>
<td>I10</td>
<td></td>
</tr>
<tr>
<td>EXE</td>
<td>I1</td>
<td>I3</td>
<td>I5</td>
<td>I7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I2</td>
<td>I4</td>
<td>I6</td>
<td>I8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>I1</td>
<td>I3</td>
<td>I5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I2</td>
<td>I4</td>
<td>I6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>I1</td>
<td>I3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>I2</td>
<td>I4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPI $\rightarrow 0.5$; IPC $\rightarrow 2$
Advanced Superscalar Execution

- Ideally: in an n-issue superscalar, n instructions are fetched, decoded, executed, and committed per cycle
- In practice:
  - Data, control, and structural hazards spoil issue flow
  - Multi-cycle instructions spoil commit flow
- Buffers at issue (issue queue) and commit (reorder buffer) decouple these stages from the rest of the pipeline and regularize somewhat breaks in the flow
Problems At Instruction Fetch

- Crossing instruction cache line boundaries
  - e.g., 32 bit instructions and 32 byte instruction cache lines \(\rightarrow\) 8 instructions per cache line; 4-wide superscalar processor

Case 1: all instructions located in same cache line and no branch

Case 2: instructions spread in more lines and no branch

  - More than one cache lookup is required in the same cycle
  - Words from different lines must be ordered and packed into instruction queue
Problems At Instruction Fetch

- **Control flow**
  - e.g., 32 bit instructions and 32 byte instruction cache lines → 8 instructions per cache line; 4-wide superscalar processor

Case 1: single not taken branch

Case 2: single taken branch outside fetch range and into other cache line

- Branch prediction is required within the instruction fetch stage
- For wider issue processors multiple predictions are likely required
- In practice most fetch units only fetch up to the first predicted taken branch
# Example Frequencies of Control Flow

<table>
<thead>
<tr>
<th>benchmark</th>
<th>taken %</th>
<th>avg. BB size</th>
<th># of inst. between taken branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>eqntott</td>
<td>86.2</td>
<td>4.20</td>
<td>4.87</td>
</tr>
<tr>
<td>espresso</td>
<td>63.8</td>
<td>4.24</td>
<td>6.65</td>
</tr>
<tr>
<td>xlisp</td>
<td>64.7</td>
<td>4.34</td>
<td>6.70</td>
</tr>
<tr>
<td>gcc</td>
<td>67.6</td>
<td>4.65</td>
<td>6.88</td>
</tr>
<tr>
<td>sc</td>
<td>70.2</td>
<td>4.71</td>
<td>6.71</td>
</tr>
<tr>
<td>compress</td>
<td>60.9</td>
<td>5.39</td>
<td>8.85</td>
</tr>
</tbody>
</table>

Data from Rotenberg et. al. for SPEC 92 Int

- One branch about every 4 to 6 instructions
- One taken branch about every 5 to 9 instructions
Solutions For Instruction Fetch

- Advanced fetch engines that can perform multiple cache line lookups
  - E.g., interleaved I-caches where consecutive program lines are stored in different banks that be can accessed in parallel
- Very fast, albeit not very accurate branch predictors (e.g. branch target buffers)
  - Note: usually used in conjunction with more accurate but slower predictors
- Restructuring instruction storage to keep commonly consecutive instructions together (e.g., Trace cache in Pentium 4)
Example Advanced Fetch Unit

Control flow prediction units:

i) Branch Target Buffer
ii) Return Address Stack
iii) Branch Predictor

Mask to select instructions from each of the cache lines

2-way interleaved I-cache

Figure from Rotenberg et. al.

Final alignment unit
Trace Caches

- Traditional I-cache: instructions laid out in program order
- Dynamic execution order does not always follow program order (e.g., taken branches) and the dynamic order also changes
- Idea:
  - Store instructions in execution order (**traces**)
  - Traces can start with any static instruction and are identified by the starting instruction’s PC
  - Traces are dynamically created as instructions are normally fetched and branches are resolved
  - Traces also contain the outcomes of the **implicitly predicted** branches
  - When the same trace is again encountered (i.e., same starting instruction and same branch predictions) instructions are obtained from trace cache
  - Note that multiple traces can be stored with the same starting instruction
Branch Prediction

- We already saw BTB for quick predictions
- Combining Predictor
  - Processors have multiple branch predictors with accuracy delay tradeoffs
  - Meta-predictor chooses what predictor to use
- Perceptron predictor
  - Uses neural-networks for branch prediction
- TAGE predictor
  - Similar to combining predictor idea but with no meta predictor
Superscalar: Other Challenges

- Superscalar decode
  - Replicate decoders (ok)

- Superscalar issue
  - Number of dependence tests increases quadratically (bad)

- Superscalar register read
  - Number of register ports increases linearly (bad)
Superscalar: Other Challenges

- **Superscalar execute**
  - Replicate functional units (Not bad)

- **Superscalar bypass/forwarding**
  - Increases quadratically (bad)
  - Clustering mitigates this problem

- **Superscalar register-writeback**
  - Increases linearly (bad)

- **ILP uncovered**
  - Limited by ILP inherent in program
  - Bigger instruction windows
Effect of Instruction Window

Instructions Per Clock

- Infinite
- 2048
- 512
- 128
- 32
References and Further Reading

- **Original hardware trace cache:**
  

- **Next trace prediction for trace caches:**
  

- **A Software trace cache:**
  
References and Further Reading

- Seminal branch prediction work:

- Neural net based branch predictors:
  Neural net based branch predictors:

- TAGE predictor

- Championship Branch Prediction
  - [www.jilp.org/cbp/](http://www.jilp.org/cbp/)
  - [http://taco.cs.utsa.edu/camino/cbp2/](http://taco.cs.utsa.edu/camino/cbp2/)
Probing Further

- **Advanced register allocation and de-allocation**

- **Value prediction**

- **Limitations to wide issue processors**
Pros/Cons of Trace Caches

+ Instructions come from a single trace cache line
+ Branches are implicitly predicted
  - The instruction that follows the branch is fixed in the trace and implies the branch’s direction (taken or not taken)
+ I-cache still present, so no need to change cache hierarchy
+ In CISC ISA’s (e.g., x86) the trace cache can keep decoded instructions (e.g., Pentium 4)
  - Wasted storage as instructions appear in both I-cache and trace cache, and in possibly multiple trace cache lines
  - Not very good when there are traces with common sub-paths
  - Not very good at handling indirect jumps and returns (which have multiple targets, instead of only taken/not taken)
Structure of a Trace Cache

Figure from Rotenberg et. al.
Structure of a Trace Cache

- Each line contains $n$ instructions from up to $m$ basic blocks
- Control bits:
  - Valid
  - Tag
  - Branch flags and mask: $m-1$ bits to specify the direction of the up to $m$ branches
  - Branch mask: the number of branches in the trace
  - Trace target address and fall-through address: the address of the next instruction to be fetched after the trace is exhausted
- Trace cache hit:
  - Tag must match
  - Branch predictions must match the branch flags for all branches in the trace
Trace Creation

- Starts on a trace cache miss
- Instructions are fetched up to the first predicted taken branch
- Instructions are collected, possibly from multiple basic blocks (when branches are predicted taken)
- Trace is terminated when either $n$ instructions or $m$ branches have been added
- Trace target/fall-through address are computed at the end
Example

- I-cache lines contain 8, 32-bit instructions and Trace Cache lines contain up to 24 instructions and 3 branches
- Processor can fetch up to 4 instructions per cycle

Machine Code

L1: I1 [ALU]
... I5 [Cond. Br. to L3]
L2: I6 [ALU]
... I12 [Jump to L4]
L3: I13 [ALU]
... I18 [Cond. Br. to L5]
L4: I19 [ALU]
... I24 [Cond. Br. to L1]
L5: 

Basic Blocks

B1 (I1-I15)
B2 (I6-I12)
B3 (I13-I18)
B4 (I19-I24)

Layout in I-Cache

B1: I1  [ALU]
...  I5  [Cond. Br. to L3]
L2: I6  [ALU]
...  I12 [Jump to L4]
L3: I13 [ALU]
...  I18 [Cond. Br. to L5]
L4: I19 [ALU]
...  I24 [Cond. Br. to L1]
Example

- Step 1: fetch I1-I3 (stop at end of line) → Trace Cache miss → Start trace collection
- Step 2: fetch I4-I5 (possible I-cache miss) (stop at predicted taken branch)
- Step 3: fetch I13-16 (possible I-cache miss)
- Step 4: fetch I17-I19 (I18 is predicted not taken branch, stop at end of line)
- Step 5: fetch I20-I23 (possible I-cache miss)
- Step 6: fetch I24 (stop at predicted taken branch)
- Step 7: fetch I1-I4 replaced by Trace Cache access

Basic Blocks

- B1 (I1-I5)
- B2 (I6-I12)
- B3 (I13-I18)
- B4 (I19-I24)

Layout in I-Cache

<table>
<thead>
<tr>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
<th>I5</th>
<th>I6</th>
<th>I7</th>
<th>I8</th>
<th>I9</th>
<th>I10</th>
<th>I11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Layout in Trace Cache

<table>
<thead>
<tr>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
<th>I5</th>
<th>I13</th>
<th>I14</th>
<th>I15</th>
<th>I16</th>
<th>I17</th>
<th>I18</th>
<th>I19</th>
<th>I20</th>
<th>I21</th>
<th>I22</th>
<th>I23</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Common path