Recap: Pipeline Performance

- Exploit *implicit* instruction-level parallelism with more complex pipelines and dynamic scheduling
- Execute instructions out-of-order while preserving:
  - True dependences (RAW)
  - Precise exceptions
- Register renaming removes WAR and WAW hazards
- Reorder buffer holds completed results before committing to support precise exceptions
- Branches are frequent and limit achievable performance due to control hazards
Recap: Overall Pipeline Structure

- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- Commit (write-back to architectural state, i.e., regfile & memory) is in-order

Temporary storage needed to hold results before commit (shadow registers and store buffers)

Control Flow Penalty

Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution!

How much work is lost if pipeline doesn’t follow correct instruction flow?

~ Loop length x pipeline width
MIPS Branches and Jumps

Each instruction fetch depends on one or two pieces of information from the preceding instruction:

1) Is the preceding instruction a taken branch?
2) If so, what is the target address?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Taken known?</th>
<th>Target known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>After Inst. Decode</td>
<td>After Inst. Decode</td>
</tr>
<tr>
<td>JR</td>
<td>After Inst. Decode</td>
<td>After Reg. Fetch</td>
</tr>
<tr>
<td>BEQZ/BNEZ</td>
<td>After Reg. Fetch*</td>
<td>After Inst. Decode</td>
</tr>
</tbody>
</table>

*Assuming zero detect on register read

Branch Penalties in Modern Pipelines

UltraSPARC-III instruction fetch pipeline stages (in-order issue, 4-way superscalar, 750MHz, 2000)

Branch Target Address Known
Branch Direction & Jump Register Target Known

PC Generation/Mux
Instruction Fetch Stage 1
Instruction Fetch Stage 2
Branch Address Calc/Begin Decode
Complete Decode
Steer Instructions to Functional units
Register File Read
Integer Execute

Remainder of execute pipeline (+ another 6 stages)
Reducing Control Flow Penalty

Software solutions
- *Eliminate branches - loop unrolling*
  Increases the run length
- *Reduce resolution time - instruction scheduling*
  Compute the branch condition as early as possible (of limited value)

Hardware solutions
- Find something else to do - *delay slots*
  Replaces pipeline bubbles with useful work (requires software cooperation)
- *Speculate - branch prediction*
  Speculative execution of instructions beyond the branch

Branch Prediction

*Motivation:*
Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

*Required hardware support:*
*Prediction structures:*
- Branch history tables, branch target buffers, etc.

*Mispredict recovery mechanisms:*
- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to state following branch
**Static Branch Prediction**

Overall probability a branch is taken is ~60-70% but:

- **backward** 90%
- **forward** 50%

ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
  - `bne0` *(preferred taken)*  `beq0` *(not taken)*

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
typically reported as ~80% accurate

9/13/2007

---

**Dynamic Branch Prediction**

*learning based on past behavior*

**Temporal correlation**

The way a branch resolves may be a good predictor of the way it will resolve at the next execution

**Spatial correlation**

Several branches may resolve in a highly correlated manner *(a preferred path of execution)*
**Branch Prediction Bits**

- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!

\[ \text{BP state: } (\text{predict take/\neg take}) \times (\text{last prediction right/wrong}) \]

---

**Branch History Table**

- Fetch PC
- I-Cache
- Opcode Offset
- Instruction
- BHT Index
- \(2^k\)-entry BHT, 2 bits/entry
- Taken/\neg Taken?

4K-entry BHT, 2 bits/entry, \sim 80-90\% correct predictions
Exploiting Spatial Correlation
Yeh and Patt, 1992

\[
\text{if } (x[i] < 7) \text{ then } \quad y += 1; \\
\text{if } (x[i] < 5) \text{ then } \quad c -= 4;
\]

If first condition false, second condition also false

*History register*, H, records the direction of the last N branches executed by the processor

Two-Level Branch Predictor

*Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)*
Limitations of BHTs

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.

Correctly predicted taken branch penalty

Jump Register penalty

Branch Target Buffer

BP bits are stored with the predicted target address.

IF stage: If (BP=taken) then nPC=target else nPC=PC+4
later: check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb
**Address Collisions**

Assume a 128-entry BTB

What will be fetched after the instruction at 1028?
- BTB prediction = 236
- Correct target = 1032

⇒ *kill* PC=236 and *fetch* PC=1032

*Is this a common occurrence? Can we avoid these bubbles?*

**BTB is only for Control Instructions**

BTB contains useful information for branch and jump instructions only

⇒ Do not update it for other instructions

For all other instructions the next PC is PC+4!

*How to achieve this effect without decoding the instruction?*
Branch Target Buffer (BTB)

- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

Consulting BTB Before Decoding

- The match for PC=1028 fails and 1028+4 is fetched
  \[\Rightarrow \text{eliminates false predictions after ALU instructions}\]
- BTB contains entries only for control transfer instructions
  \[\Rightarrow \text{more room to store branch targets}\]
Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate

BHT in later pipeline stage corrects when BTB misses a predicted taken branch

<table>
<thead>
<tr>
<th>BTB</th>
<th>BHT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>PC Generation/Mux</td>
</tr>
<tr>
<td>P</td>
<td>Instruction Fetch Stage 1</td>
</tr>
<tr>
<td>F</td>
<td>Instruction Fetch Stage 2</td>
</tr>
<tr>
<td>B</td>
<td>Branch Address Calc/Begin Decode</td>
</tr>
<tr>
<td>I</td>
<td>Complete Decode</td>
</tr>
<tr>
<td>J</td>
<td>Steer Instructions to Functional units</td>
</tr>
<tr>
<td>R</td>
<td>Register File Read</td>
</tr>
<tr>
<td>E</td>
<td>Integer Execute</td>
</tr>
</tbody>
</table>

BTB/BHT only updated after branch resolves in E stage

9/13/2007 21

Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)
  
  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)
  
  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)
  
  BTB works well if usually return to the same place
  ⇒ Often one function called from many distinct call sites!

How well does BTB work for each of these cases?

9/13/2007 22
**Subroutine Return Stack**

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```c
fa() { fb(); }
fb() { fc(); }
fC() { fd(); }
```

**Mispredict Recovery**

In-order execution machines:
- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution?
- Multiple instructions following branch in program order can complete before branch resolves
**In-Order Commit for Precise Exceptions**

- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- *Commit* (write-back to architectural state, i.e., regfile & memory, is in-order)

*Temporary storage needed in ROB to hold results before commit*

---

**Branch Misprediction in Pipeline**

- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch
Speculating Both Directions

An alternative to branch prediction is to execute both directions of a branch *speculatively*

- resource requirement is proportional to the number of concurrent speculative executions
- only half the resources engage in useful work when both directions of a branch are executed speculatively
- branch prediction takes less resources than speculative execution of both paths

*With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction*
CS252 Administrivia

- Prereq quiz, hand back at end of class
- Projects, see web page over weekend
  - Benchmarking parallel programs/architectures - how?
  - Take a favorite application and parallelize it for a Multicore/GPU
  - On-chip network design using RAMP Blue/UPC/NAS benchmarks
    - Processor-network interface - design a better one
    - Network generator - design a generator for different routers/interconnects, evaluate performance on UPC benchmarks
  - Reduce power of memory system on Niagara-2, change OS/hardware regs
  - Where does memory bandwidth go? Work on finding where current machines lose ability to saturate their memory systems, suggest memory performance counters
  - Use RAMP/Leon to build a very fast simulator that captures program stats, evaluate large Linux applications
  - Never too early to come up with your own idea!
- Next reading assignment “Limits of ILP” by David Wall. Read pages 1-35 (back contains long appendices). Summarize in one page, and include descriptions of any flaws you found in study. Discuss in class on Tuesday Sep 18.

9/13/2007

“Data in ROB” Design
(HP PA8000, Pentium Pro, Core2Duo)

- On dispatch into ROB, ready sources can be in regfile or in ROB dest (copied into src1/src2 if ready before dispatch)
- On completion, write to dest field and broadcast to src fields.
- On issue, read from ROB src fields

9/13/2007
Unified Physical Register File
(MIPS R10K, Alpha 21264, Pentium 4)

- One regfile for both committed and speculative values (no data in ROB)
- During decode, instruction result allocated new physical register, source regs translated to physical reg through rename table
- Instruction reads data from regfile at start of execute (not in decode)
- Write-back updates reg. busy bits on instructions in ROB (assoc. search)
- Snapshots of rename table taken at every branch to recover mispredicts
- On exception, renaming undone in reverse order of issue (MIPS R10000)

Pipeline Design with Physical Regfile
**Lifetime of Physical Registers**

- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries (*no data in ROB*)

```plaintext
ld r1, (r3)          ld P1, (Px)  
add r3, r1, #4       add P2, P1, #4  
sub r6, r7, r9       sub P3, Py, Pz  
add r3, r3, r6       add P4, P2, P3  
ld r6, (r1)          ld P5, (P1)   
add r6, r6, r3       add P6, P5, P4  
st r6, (r1)          st P6, (P1)   
ld r6, (r11)         ld P7, (Pw)   
```

When can we reuse a physical register?

*When next write of same architectural register commits*

**Physical Register Management**

<table>
<thead>
<tr>
<th>Rename Table</th>
<th>Physical Regs</th>
<th>Free List</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>P0</td>
<td>P0</td>
</tr>
<tr>
<td>R1</td>
<td>P1</td>
<td>P1</td>
</tr>
<tr>
<td>R2</td>
<td>P2</td>
<td>P3</td>
</tr>
<tr>
<td>R3</td>
<td>P3</td>
<td>P2</td>
</tr>
<tr>
<td>R4</td>
<td>P4</td>
<td>P4</td>
</tr>
<tr>
<td>R5</td>
<td>P5</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>P6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(LPRd requires third read port on Rename Table for each instruction)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ROB</th>
</tr>
</thead>
<tbody>
<tr>
<td>use</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

9/13/2007 33
9/13/2007 34
Physical Register Management

<table>
<thead>
<tr>
<th>Rename Table</th>
<th>Physical Regs</th>
<th>Free List</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>P0</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>P7</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>P5</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td>P6</td>
<td></td>
</tr>
</tbody>
</table>

ROB

<table>
<thead>
<tr>
<th>use</th>
<th>ex</th>
<th>op</th>
<th>p1</th>
<th>PR1</th>
<th>p2</th>
<th>PR2</th>
<th>Rd</th>
<th>LPRd</th>
<th>PRd</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>P7</td>
<td></td>
<td></td>
<td>r1</td>
<td>P8</td>
<td>P0</td>
</tr>
</tbody>
</table>

ld r1, 0(r3)  
add r3, r1, #4  
sub r6, r7, r6  
add r3, r3, r6  
ld r6, 0(r1)
### Physical Register Management

#### Rename Table

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>P0</td>
<td>P1</td>
<td>P1</td>
<td>P2</td>
<td>P2</td>
<td>P3</td>
<td>P3</td>
</tr>
</tbody>
</table>

#### Physical Regs

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;R6&gt;</td>
<td>&lt;R7&gt;</td>
<td>&lt;R3&gt;</td>
</tr>
</tbody>
</table>

#### Free List

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>P4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### ROB

<table>
<thead>
<tr>
<th>use</th>
<th>ex</th>
<th>op</th>
<th>p1</th>
<th>PR1</th>
<th>p2</th>
<th>PR2</th>
<th>Rd</th>
<th>LPRd</th>
<th>PRd</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>Id</td>
<td>p</td>
<td>P7</td>
<td>P2</td>
<td>r1</td>
<td>P8</td>
<td>r0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>add</td>
<td>P0</td>
<td>r3</td>
<td>P7</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>sub</td>
<td>P6</td>
<td>r5</td>
<td>P5</td>
<td>r6</td>
<td>P5</td>
<td>P3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Id r1, 0(r3)
- add r3, r1, #4
- sub r6, r7, r6
- add r3, r3, r6
- Id r6, 0(r1)
Physical Register Management

Renamed Table

| R0 | P0 |
| R1 | P0 |
| R2 | P2 |
| R3 | P2 |
| R4 | P4 |
| R5 | P4 |
| R6 | P6 |
| R7 | P6 |

Physical Regs

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;R6&gt;</td>
<td></td>
<td>&lt;R7&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;R3&gt;</td>
<td></td>
<td>&lt;R1&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Free List

id r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6

ld r6, 0(r1)

ROB

<table>
<thead>
<tr>
<th>use</th>
<th>ex</th>
<th>op</th>
<th>p1</th>
<th>PR1</th>
<th>p2</th>
<th>PR2</th>
<th>Rd</th>
<th>LPRd</th>
<th>PRd</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>id</td>
<td>p</td>
<td>P7</td>
<td></td>
<td>r1</td>
<td>P8</td>
<td>P0</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>add</td>
<td>P0</td>
<td></td>
<td>P6</td>
<td></td>
<td>P5</td>
<td>r6</td>
<td>P5</td>
</tr>
<tr>
<td>x</td>
<td>sub</td>
<td>P7</td>
<td></td>
<td>P5</td>
<td>r6</td>
<td>P5</td>
<td>P3</td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>id</td>
<td>P0</td>
<td></td>
<td>r6</td>
<td>P3</td>
<td>P2</td>
<td>P4</td>
<td></td>
</tr>
</tbody>
</table>

Execute & Commit

ld r1, 0(r3)
add r3, r1, #4
sub r6, r7, r6
add r3, r3, r6

ld r6, 0(r1)
### Physical Register Management

#### Rename Table

<table>
<thead>
<tr>
<th>ROB</th>
<th>PR0</th>
<th>PR1</th>
<th>PR2</th>
<th>PR3</th>
<th>PR4</th>
<th>PR5</th>
<th>PR6</th>
<th>PR7</th>
<th>PR8</th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td>use</td>
<td>ex</td>
<td>op</td>
<td>p1</td>
<td>PR1</td>
<td>p2</td>
<td>PR2</td>
<td>Rd</td>
<td>LPRd</td>
<td>PRd</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>p</td>
<td>P7</td>
<td>r1</td>
<td>P2</td>
<td>r3</td>
<td>P8</td>
<td>P0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>p</td>
<td>P0</td>
<td>r3</td>
<td>P7</td>
<td>r3</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>sub</td>
<td>p</td>
<td>P6</td>
<td>r3</td>
<td>P5</td>
<td>r3</td>
<td>P3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>add</td>
<td>p</td>
<td>P1</td>
<td>r3</td>
<td>P3</td>
<td>r3</td>
<td>P2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>ld</td>
<td>p</td>
<td>P0</td>
<td>r6</td>
<td>P3</td>
<td>r6</td>
<td>P4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Physical Regs

- ld r1, 0(r3)
- add r3, r1, #4
- sub r6, r7, r9
- add r3, r3, r6
- ld r6, 0(r1)

#### Free List

### Reorder Buffer Holds

#### Active Instruction Window

- **(Older instructions)**
  - ld r1, (r3)
  - add r3, r1, r2
  - sub r6, r7, r9
  - add r3, r3, r6
  - ld r6, (r1)
  - add r6, r6, r3
  - st r6, (r1)
  - ld r6, (r1)

- **(Newer instructions)**

- **Cycle t**
  - Commit
  - Execute
  - Fetch

- **Cycle t + 1**
Superscalar Register Renaming

- During decode, instructions allocated new physical destination register
- Source operands renamed to physical register with newest value
- Execution unit only sees physical register numbers

Inst 1
\[\text{Op} \mid \text{Dest} \mid \text{Src1} \mid \text{Src2}\]

Inst 2
\[\text{Op} \mid \text{Dest} \mid \text{Src1} \mid \text{Src2}\]

Update Mapping

Read Addresses

Rename Table

Read Data

Register Free List

Op | PDest | PSrc1 | PSrc2

Op | PDest | PSrc1 | PSrc2

Does this work?

Superscalar Register Renaming

Inst 1
\[\text{Op} \mid \text{Dest} \mid \text{Src1} \mid \text{Src2}\]

Inst 2
\[\text{Op} \mid \text{Dest} \mid \text{Src1} \mid \text{Src2}\]

Update Mapping

Must check for RAW hazards between instructions issuing in same cycle. Can be done in parallel with rename lookup.

\[\text{Op} \mid \text{PDest} \mid \text{PSrc1} \mid \text{PSrc2}\]

\[\text{Op} \mid \text{PDest} \mid \text{PSrc1} \mid \text{PSrc2}\]

MIPS R10K renames 4 serially-RAW-dependent insts/cycle
Memory Dependencies

```
st r1, (r2)
ld r3, (r4)
```

When can we execute the load?

In-Order Memory Queue

• Execute all loads and stores in program order

=> Load and store cannot leave ROB for execution until all previous loads and stores have completed execution

• Can still execute loads and stores speculatively, and out-of-order with respect to other instructions
**Conservative O-o-O Load Execution**

\[
\begin{align*}
st & \ r1, \ (r2) \\
ld & \ r3, \ (r4)
\end{align*}
\]

- Split execution of store instruction into two phases: address calculation and data write
- Can execute load before store, if addresses known and \( r4 \neq r2 \)
- Each load address compared with addresses of all previous uncommitted stores (can use partial conservative check i.e., bottom 12 bits of address)
- Don’t execute load if any previous store address not known

\((MIPS \ R10K, \ 16 \ entry \ address \ queue)\)

---

**Address Speculation**

\[
\begin{align*}
st & \ r1, \ (r2) \\
ld & \ r3, \ (r4)
\end{align*}
\]

- Guess that \( r4 \neq r2 \)
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find \( r4=r2 \), squash load and all following instructions

\(\Rightarrow\) Large penalty for inaccurate address speculation
Memory Dependence Prediction

(Alpha 21264)

\[
\text{st } r1, (r2) \\
\text{ld } r3, (r4)
\]

- Guess that \( r4 \neq r2 \) and execute load before store
- If later find \( r4==r2 \), squash load and all following instructions, but mark load instruction as \textit{store-wait}
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear \textit{store-wait} bits

Speculative Loads / Stores

Just like register updates, stores should not modify the memory until after the instruction is committed

- A speculative store buffer is a structure introduced to hold speculative store data.
Speculative Store Buffer

- On store execute:
  - mark entry valid and speculative, and save data and tag of instruction.
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

• If data in both store buffer and cache, which should we use:
  Speculative store buffer
• If same address in store buffer twice, which should we use:
  Youngest store older than load

9/13/2007
Paper Discussion: CISC vs RISC

Recommended optional further reading:

- D. Bhandarkar and D. W. Clark. “Performance from architecture: Comparing a RISC and a CISC with similar hardware organization”, In Intl. Conf. on Architectural Support for Prog. Lang. and Operating Sys., ASPLOS-IV, Santa Clara, CA, Apr. 1991, pages 310--319 - conclusion is RISC is 2.7x better than CISC!