Last time in Lecture 13

- Register renaming removes WAR, WAW hazards
- Instruction execution divided into four major stages:
  - Instruction Fetch, Decode/Rename, Execute/Complete, Commit
- Control hazards are serious impediment to superscalar performance
- Dynamic branch predictors can be quite accurate (>95%) and avoid most control hazards
- Branch History Tables (BHTs) just predict direction (later in pipeline)
  - Just need a few bits per entry (2 bits gives hysteresis)
  - Need to decode instruction bits to determine whether this is a branch and what the target address is
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!

*BP state:*

\[
(predict \ take/\neg\take) \times (last\ prediction\ right/wrong)
\]
Branch History Table

4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Exploiting Spatial Correlation

Yeh and Patt, 1992

if (x[i] < 7) then
  y += 1;
if (x[i] < 5) then
  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)*

![Diagram of Two-Level Branch Predictor]

**Limitations of BHTs**

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

![Diagram of UltraSPARC-III fetch pipeline]
**Branch Target Buffer**

BP bits are stored with the predicted target address.

IF stage: If \((BP=\text{taken})\) then \(nPC=\text{target}\) 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

\[ \Rightarrow \text{kill } PC=236 \text{ and fetch } PC=1032 \]

*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
  ⇒ eliminates false predictions after ALU instructions

- BTB contains entries only for control transfer instructions
  ⇒ 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

BTB/BHT only updated after branch resolves in E stage
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?

Subroutine Return Stack

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

```c
fa() {
    fb();
}

fb() {
    fc();
}

fc() {
    fd();
}
```

Push call address when function call executed

Pop return address when subroutine return decoded

k entries (typically k=8-16)
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

Recovering ROB/Renaming Table

Take snapshot of register rename table at each predicted branch, recover earlier snapshot if branch mispredicted
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*

---

CS152 Administrivia

- Quiz 3, Thursday March 19, Virtual Memory
“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

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 regs 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

- **Branch Prediction**
- **Fetch**
- **Decode & Rename**
- **Reorder Buffer**
- **Commit**
- **Update predictors**
- **Out-of-Order**
- **In-Order**
- **Physical Reg. File**
- **Branch Unit**
- **ALU**
- **MEM**
- **Store Buffer**
- **D$**
- **In-Order**

**Lifetime of Physical Registers**

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

```
Id r1, (r3)
add r3, r1, #4
sub r6, r7, r9
add r3, r3, r6
Id r6, (r1)
add r6, r6, r3
st r6, (r1)
Id r6, (r11)

Id P1, (Px)
add P2, P1, #4
sub P3, Py, Pz
add P4, P2, P3
Id P5, (P1)
add P6, P5, P4
st P6, (P1)
Id 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>&lt;R6&gt; p</td>
</tr>
<tr>
<td>R6</td>
<td>P6</td>
<td>&lt;R7&gt; p</td>
</tr>
<tr>
<td>R7</td>
<td>P7</td>
<td>&lt;R3&gt; p</td>
</tr>
<tr>
<td>R8</td>
<td>P8</td>
<td>&lt;R1&gt; p</td>
</tr>
<tr>
<td>Pn</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></td>
<td>x</td>
<td>ld</td>
<td>p</td>
<td>P7</td>
<td></td>
<td>P2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

(LPRd requires third read port on Rename Table for each instruction)
Physical Register Management

### Rename Table

<table>
<thead>
<tr>
<th>Physical Regs</th>
<th>Free List</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td></td>
</tr>
<tr>
<td>P9</td>
<td></td>
</tr>
<tr>
<td>P10</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>ld</td>
<td>p</td>
<td>P7</td>
<td></td>
<td></td>
<td>r1</td>
<td>P8</td>
<td>P0</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>add</td>
<td>P0</td>
<td>r3</td>
<td>P7</td>
<td>P1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instructions

- 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></td>
<td></td>
<td>P2</td>
<td></td>
<td></td>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td>P4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td></td>
<td>P3</td>
<td></td>
<td>P4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>P4</td>
<td></td>
<td></td>
<td></td>
<td>P6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Physical Registers (PRs)

<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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Free List

P0, P1, P2, P3, P4, P5, P6, P7, P8

---

### 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>ld</td>
<td>p</td>
<td>P7</td>
<td></td>
<td></td>
<td>r1</td>
<td>P8</td>
<td>P0</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>add</td>
<td>P0</td>
<td></td>
<td>P3</td>
<td></td>
<td>r3</td>
<td>P7</td>
<td>P1</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>sub</td>
<td>p</td>
<td>P6</td>
<td></td>
<td>P5</td>
<td>r6</td>
<td>P5</td>
<td>P3</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>add</td>
<td>P1</td>
<td></td>
<td>P3</td>
<td></td>
<td>r3</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>ld</td>
<td>P0</td>
<td></td>
<td></td>
<td></td>
<td>r6</td>
<td>P3</td>
<td>P4</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**
  - R0
  - R1
  - R2
  - R3
  - R4
  - R5
  - R6
  - R7

- **Physical Regs**
  - P0
  - P1
  - P2
  - P3
  - P4
  - P5
  - P6
  - P7
  - P8
  - Pn

- **Free List**
  - P0
  - P1
  - P2
  - P3
  - P4
  - P5
  - P6
  - P7
  - P8

- **ROB**
  - use
  - ex
  - op
  - p1
  - PR1
  - p2
  - PR2
  - Rd
  - LPRd
  - PRd
  - x
  - x
  - ld
  - p
  - P7
  - r1
  - P8
  - P0
  - x
  - add
  - p
  - P0
  - r3
  - P7
  - P1
  - x
  - sub
  - p
  - P6
  - p
  - P5
  - r6
  - P5
  - P3
  - x
  - add
  - P1
  - P3
  - r3
  - P1
  - P2
  - x
  - ld
  - P0
  - r6
  - P3
  - P4


**Execute & Commit**

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

Commit

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)

Execute

Cycle t

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

Op | Dest | Src1 | Src2
---|------|------|------
Op | Dest | Src1 | Src2

Update Mapping

Read Addresses

Rename Table

Register Free List

Write Ports

Read Data

Inst 2

Op | PDest | PSrc1 | PSrc2
---|------|------|------
Op | PDest | PSrc1 | PSrc2

Does this work?
Superscalar Register Renaming

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

Update Mapping

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

Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

• MIT material derived from course 6.823
• UCB material derived from course CS252