Last time in Lecture 13

- Register renaming removes WAR, WAW hazards by giving a new internal destination register for every new result
- Pipeline is structured with in-order fetch/decode/rename, followed by out-of-order execution/complete, followed by in-order commit
- At commit time, can detect exceptions and roll back buffer to provide precise interrupts
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

- A: PC Generation/Mux
- P: Instruction Fetch Stage 1
- F: Instruction Fetch Stage 2
- B: Branch Address Calc/Begin Decode
- I: Complete Decode
- J: Steer Instructions to Functional units
- R: Register File Read
- E: 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:

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

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!

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

**Branch History Table**

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

Yeh and Patt, 1992

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

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](image)
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

UltraSPARC-III fetch pipeline

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
  \[\Rightarrow \text{more room to store branch targets}\]
CS152 Administrivia

- Lab 4, branch predictor competition, due April 3
  - **PRIZE** (TBD) for winners in both unlimited and realistic categories
- Quiz 4, Tuesday April 8

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*

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