CSE 490/590 Computer Architecture

ILP III

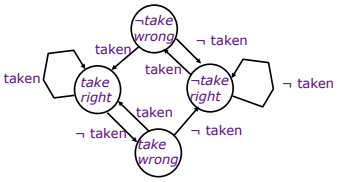
Steve Ko
Computer Sciences and Engineering
University at Buffalo

Last time…

- 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

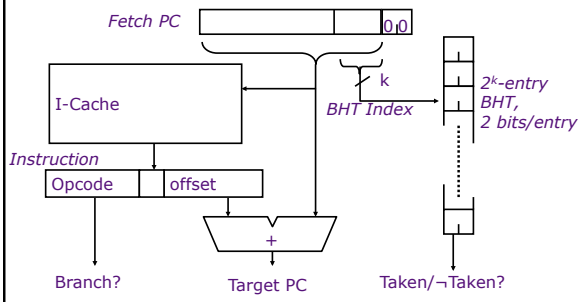
Branch Prediction Bits

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



BP state:
(*predict* take/¬take) x (*last prediction* right/wrong)

Branch History Table



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

Exploiting Spatial Correlation

Yeh and Patt, 1992

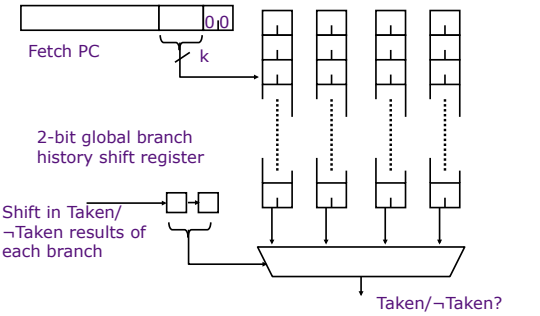
```
if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;
```

If first condition false, second condition also false

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

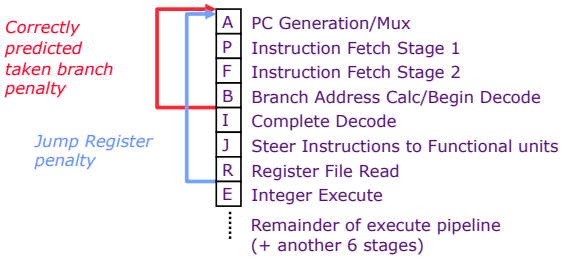
Two-Level Branch Predictor

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



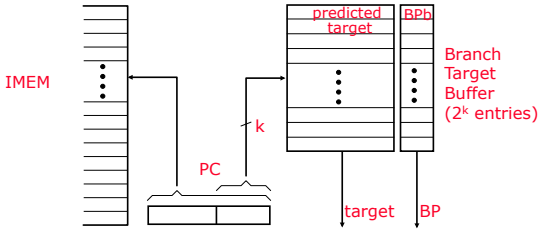
Limitations of BHTs

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



UltraSPARC-III fetch pipeline

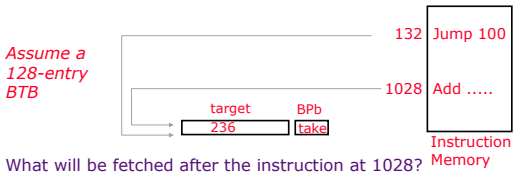
Branch Target Buffer



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



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?

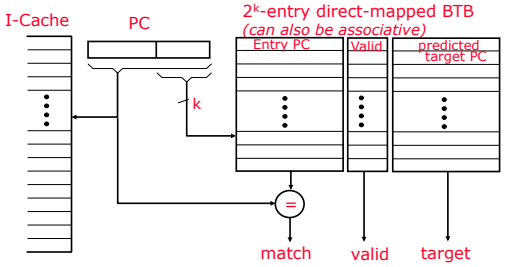
BTB is only for Control Instructions

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?

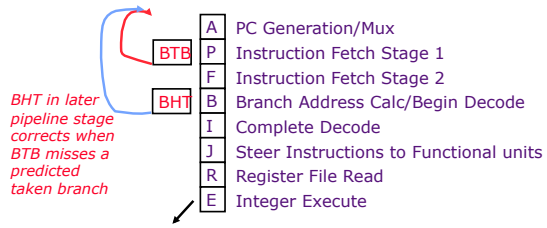
Branch Target Buffer (BTB)



- 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

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



BTB/BHT only updated after branch resolves in E stage

CSE 490/590 Administrivia

- Project 1 & midterm on Friday
  - Regrading -> Jangyoung

Mispredict Recovery

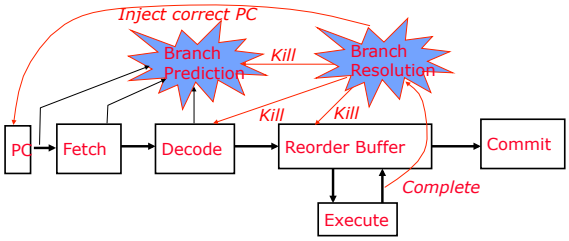
In-order execution machines:

- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution?

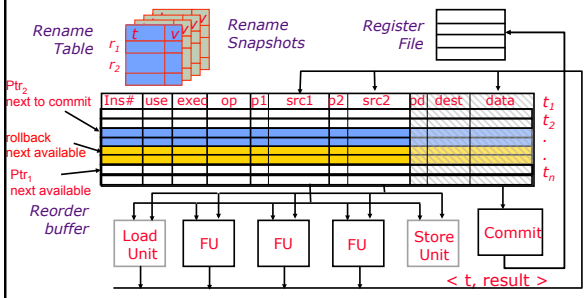
- Multiple instructions following branch in program order can complete before branch resolves

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

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
- Need a structure to handle memory ordering…

Conservative O-o-O Load Execution

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

- Split execution of store instruction into two phases: address calculation and data write
- Can execute load before store, if addresses known and r4 != 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

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

- Guess that r4 != 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

=> Large penalty for inaccurate address speculation

Memory Dependence Prediction
(Alpha 21264)

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

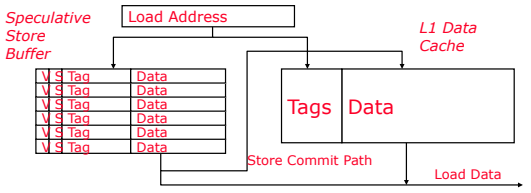
- Guess that r4 != r2 and execute load before store
- If later find r4==r2, squash load and all following instructions, but mark load instruction as *store-wait*
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear *store-wait* bits

Speculative Loads / Stores

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

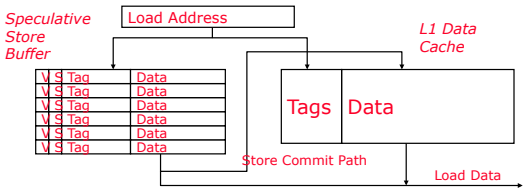
- A speculative store buffer is a structure introduced to hold speculative store data.

Speculative Store Buffer



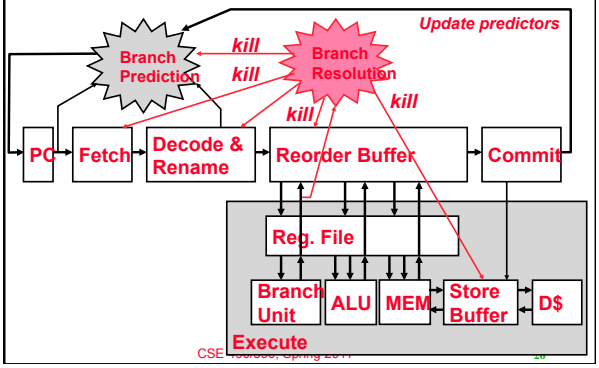
- On store execute:
  - mark entry valid and speculative, and save data and tag of instruction.
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

Speculative Store Buffer



- 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

Datapath: Branch Prediction and Speculative Execution



Acknowledgements

- These slides heavily contain material developed and copyright by
  - Krste Asanovic (MIT/UCB)
  - David Patterson (UCB)
- And also by:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252