EECS 370

Control Hazards and Performance

Classic performance problem

- Program with following instruction breakdown:
  - lw 10%
  - sw 15%
  - beq 25%
  - R-type 50%
- Speculate "always not-taken" and squash. 80% of branches not-taken
- Full forwarding to execute stage. 20% of loads stall for 1 cycle
- What is the CPI of the program?
- What is the total execution time per instruction if clock frequency is 100MHz?

CPI = 1 + 0.10 (loads) * 0.20 (load use stall)*1
+ 0.25 (branch) * 0.20 (miss rate)*3
CPI = 1 + 0.02 + 0.15 = 1.17
Time = 1.17 * 10ns =11.7ns per instruction

Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- Practice Performance Problems
  - Problem 1
  - **Problem 2**
  - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor

Classic performance problem (cont.)

- Assume branches are resolved at Execute?
  - What is the CPI?
  - What happens to cycle time?

CPI = 1 + 0.10 (loads) *0.20 (load use stall)*1
+ 0.25 (branch) * 0.20 (miss rate)*2
CPI = 1 + 0.02 + 0.1 = 1.12

Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- Practice Performance Problems
  - Problem 1
  - Problem 2
  - **Problem 3**
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- Improving Direction Predictor

Performance with deeper pipelines

- Assume the setup of the previous problem.
- What if we have a 10 stage pipeline?
  - Instructions are fetched at stage 1.
  - Register file is read at stage 3.
  - Execution begins at stage 5.
  - Branches are resolved at stage 7.
  - Memory access is complete in stage 9.
- What's the CPI of the program?
- If the clock rate was doubled by doubling the pipeline depth, is performance also doubled?

CPI = 1 + 0.10 (loads) *0.20 (load use stall)*4 + 0.25 (branch) * 0.20 (N stalls)*6
CPI = 1 + 0.08 + 0.30 = 1.38
Time = 1.38 * 5ns = 6.9 ns per instruction

Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- Practice Performance Problems
  - Problem 1
  - Problem 2
  - Problem 3
- **Improving Performance with Branch Predicting**
- Simple Direction Predictor
- Improving Direction Predictor

Can We Improve Branch Performance?

- CPI increases every time a branch is taken!
  - About 50%-66% of time
- Is that necessary?

Poll: If you had to guess, in real programs, what's the ratio of taken to not-taken branches?

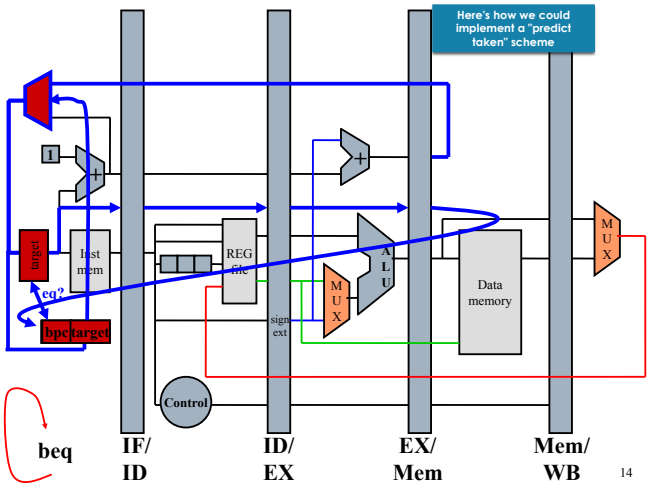
a) Very rarely taken
b) Slightly biased towards not taken
c) Slightly biased towards taken
d) Very rarely not taken

- **No!** We can try to predict when branch is taken
  - But we would need to send target PC to memory before decoding branch
  - How do we:
    1. Know an instruction is a branch before decoding?
    2. Reliably guess whether it should be taken?
    3. Figure out the target PC before executing the branch?

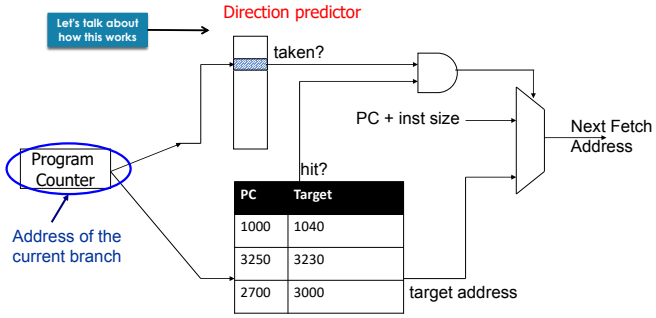
Sometimes predict taken?

- When fetching an instruction, need to predict 3 things:
  1. Whether the fetched instruction is a branch
  2. Branch direction (if conditional)
  3. Branch target address (if direction is taken)

- Observation: Target address remains the same for conditional branch across multiple executions
  - Idea: store the target address of branch once we execute it, along with PC of instruction
  - Called Branch Target Buffer (BTB)



Sometimes predict taken?



"Cache" of Target Addresses (BTB: Branch Target Buffer)

Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- Practice Performance Problems
  - Problem 1
  - Problem 2
  - Problem 3
- Improving Performance with Branch Predicting
- **Simple Direction Predictor**
- Improving Direction Predictor

Branch Direction Prediction

Poll: Overall, which do you expect to be more accurate?

- "Branch direction" refers to whether the branch was taken or not
- Two methods for predicting direction:
  - Static - We predict once during compilation, and that prediction never changes
  - Dynamic - We predict (potentially) many times during execution, and the prediction may change over time

- *Static vs dynamic strategies are a very common topic in computer architecture*

Branch Direction Prediction (Static)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40%
  - Compiler can layout code such that the likely path is the "not-taken" path

- Always taken
  - No direction prediction
  - Better accuracy: ~60-70%
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC

- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

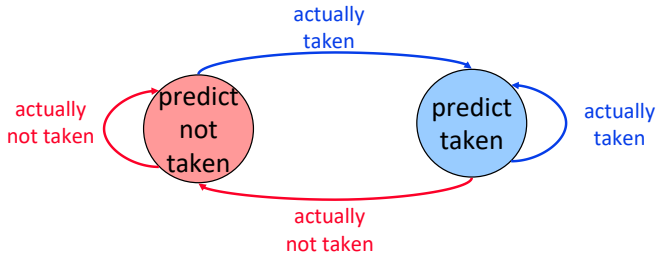
Branch Direction Prediction (Dynamic)

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed
    TTTTTTTTTNNNNNNNNNN → 90% accuracy

- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

+ Loop branches for loops with large number of iterations
-- Loop branches for loops will small number of iterations
TNTNTNTNTNTNTNTNTN → 0% accuracy

State Machine for Last-Time Prediction



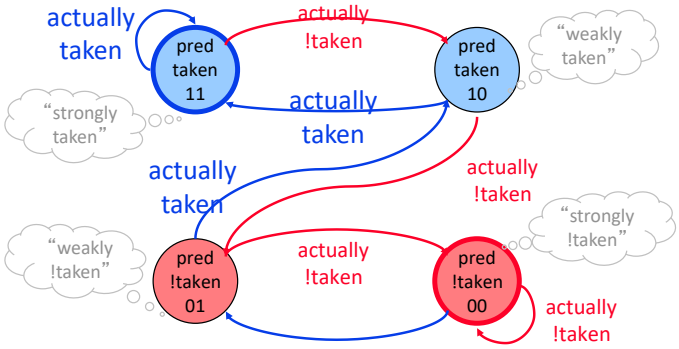
Agenda

- Control Hazards and Basic Approaches
- Detect-and-Stall
- Speculate-and-Squash
- Exceptions
- Practice Performance Problems
  - Problem 1
  - Problem 2
  - Problem 3
- Improving Performance with Branch Predicting
- Simple Direction Predictor
- **Improving Direction Predictor**

Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - Even though the branch may be mostly taken or mostly not taken
- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each

State Machine for 2-bit Saturating Counter



Two-Bit Counter Based Prediction

Poll: How many branches do we get wrong?

- What's the prediction accuracy of a branch with the following sequence of taken/not taken outcomes:
  - T T T T N T T N N N T N T N N

Br	T	T	T	T	N	T	T	N	N	N	T	N	T	N	N
State	10	11	11	11	10	11	10	11	10	01	10	01	10	01	00
Pred	T	T	T	T	T	T	T	T	T	N	N	N	N	N	N

Live Poll + Q&A: slido.com #eecs370

Can We Do Better?

- Absolutely… take 470
  - Tons of sophisticated branch predictor designs
- I've worked on a few that found their way into some Chromebooks!

Branch Prediction

- Predict not taken: ~50% accurate
- Predict backward taken: ~65% accurate
- Predict same as last time: ~80% accurate
- Realistic designs: ~96% accurate

Remember this Example from Lecture 1?

- We know understand why sorting improves the inner-loop so much
  - The branch predictor is better at guessing what's gonna happen when data is sorted!

```
for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;
std::sort(data, data + arraySize);

// Test
clock_t start = clock();
long long sum = 0;
// Primary loop
for (unsigned c = 0; c < arraySize; ++c)
{
    if (data[c] >= 128)
        sum += data[c];
}

double elapsedTime =
  static_cast<double>(clock() - start);
```

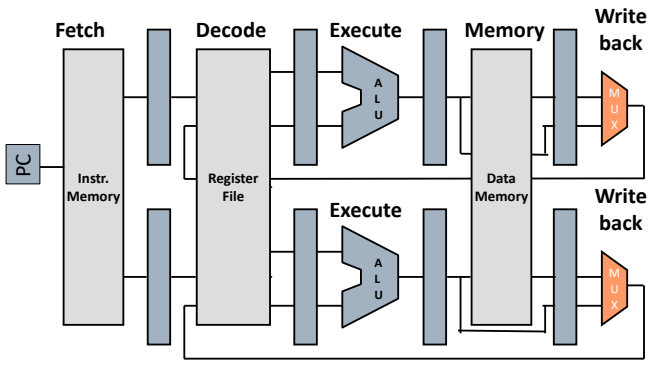

Advanced Pipelining

Not on the exam.

Creating more pipelines

- Instruction Level Parallelism – Superscalar Pipeline
  - Have two or more pipelines in same processor
  - pipelines need to work in tandem to improve single program performance
- Thread Level Parallelism – Multi-core
  - Have two or more processors (Independent Pipelines)
  - Need more programs or a parallel program
  - does not improve single program performance
- Data Level Parallelism – Single Inst. Multiple Data (SIMD)
  - Have two or more execution pipelines (ID->WB)
  - Share the same fetch and control pipeline to save power (IF+cont.)
  - Similar to GPU's

ILP Techniques: Superscalar



Other Techniques for ILP: Out of Order Execution

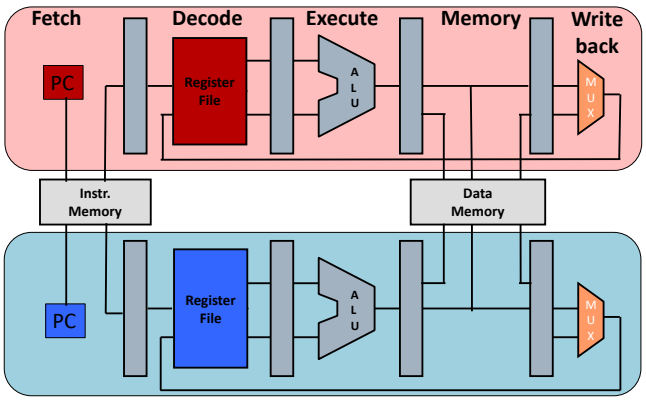
- Eliminating stall conditions decreases CPI
- Reorder instructions to avoid stalls
- Example (5-stage LC2K pipeline):

```
add   1 2 3          add   1 2 3
lw    3 2 16         lw    3 2 16
nor   2 6 7    ---->  add   4 5 1
add   4 5 1    ---->  nor   2 6 7
```

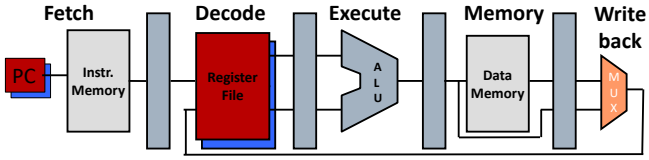

Why Use Out of Order Execution?

- Some instructions take a long time to execute
  - Floating point operations
  - Some loads and stores (more when we talk about memory hierarchy)
- Options:
  - Increase cycle time
  - Increase number of pipeline stages
  - Execute other instructions while you wait

TLP Techniques: Multiprocessors

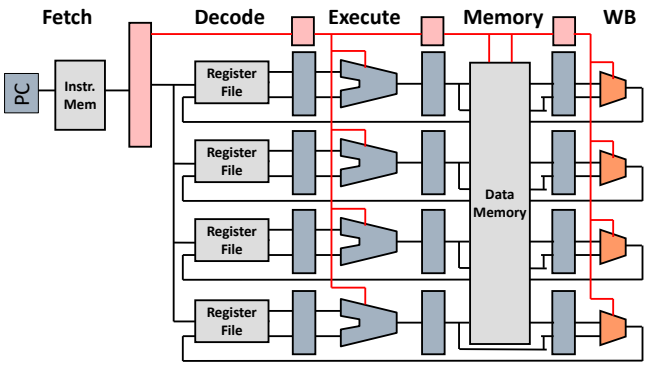


Other Techniques for TLP: Multi-Threading

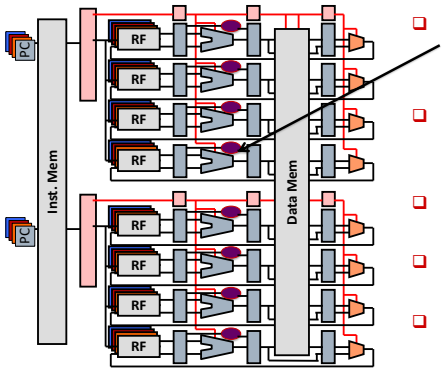


- Virtual Multiprocessor (Multi-Threading or HyperThreading)
  - Duplicate the state (PC, Registers) but time share hardware
  - User/Operating system see 2 cores, but only one execution
  - Used to hide long latencies (i.e. memory access to disk)

DLP Techniques: Single Instr. Multiple Data (SIMD)



Building a GPU



- Add special functional units in EX
- Combine Techniques
  - SIMD + MP + MT= SIMT
- MT used to hide memory latencies
- SIMD used to decrease power of fetch/control
- MP used to improve throughput