16.482 / 16.561 Computer Architecture and Design

Instructor: Dr. Michael Geiger

Spring 2014

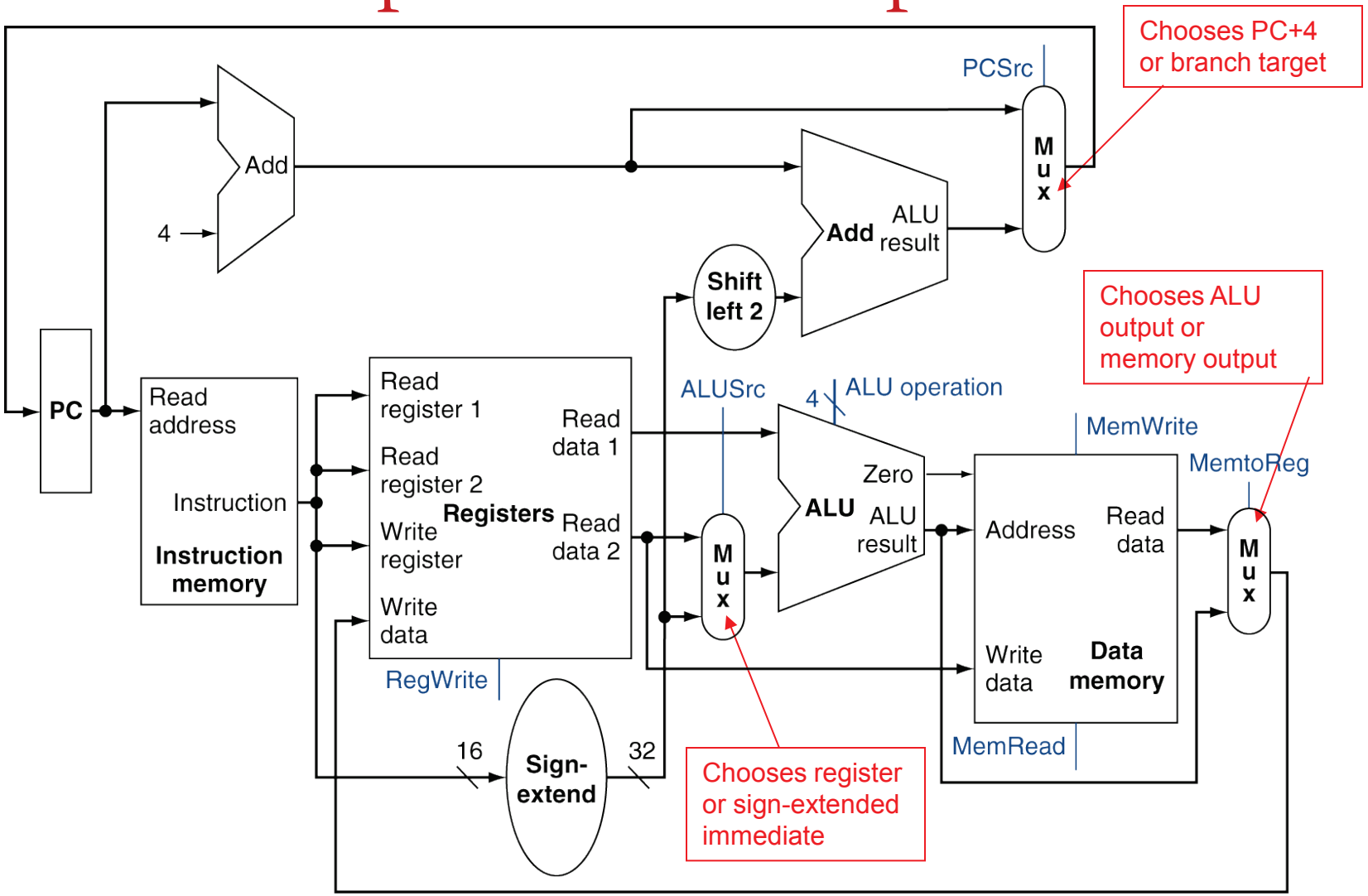
Lecture 4:

ILP and branch prediction

Lecture outline

- Announcements/reminders
  - HW 2 due today
  - HW 3 to be posted; due 2/27
- Review
  - Basic datapath design
  - Single-cycle datapath
  - Pipelining
- Today's lecture
  - Instruction level parallelism (intro)
  - Dynamic branch prediction

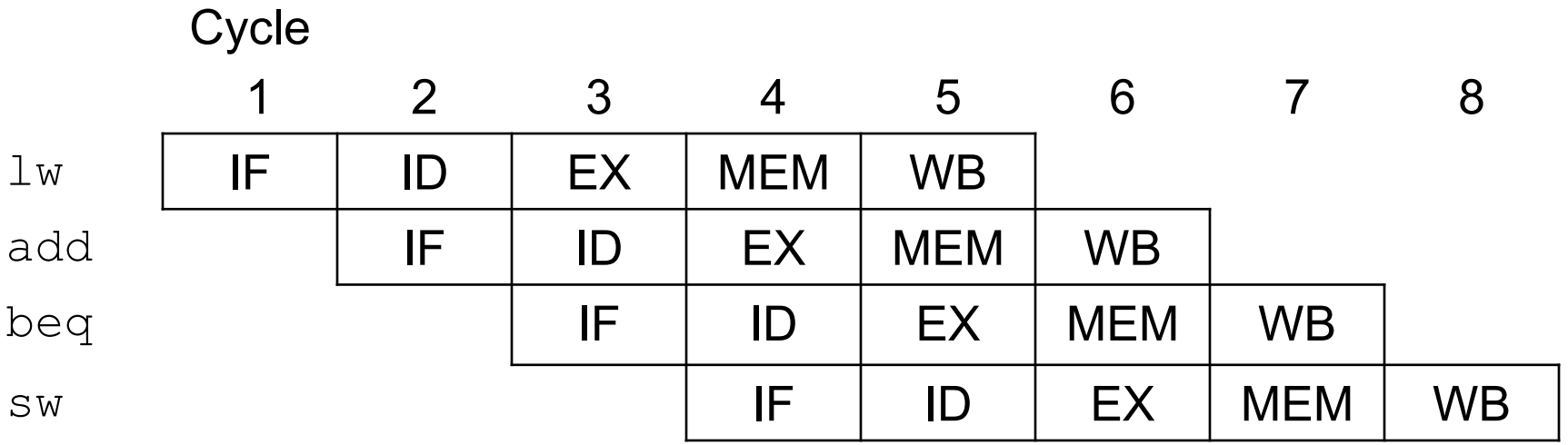
Review: Simple MIPS datapath



Review: Pipelining

- Pipelining → low CPI and a short cycle
  - Simultaneously execute multiple instructions
  - Use multi-cycle "assembly line" approach
  - Use staging registers between cycles to hold information
- Hazards: situation that prevents instruction from executing during a particular cycle
  - Structural hazards: hardware conflicts
  - Data hazards: dependences cause instruction stalls; can resolve using:
    - No-ops: compiler inserts stall cycles
    - Forwarding: add hardware paths to ALU inputs
  - Control hazards: must wait for branches
    - Can move target, comparison into ID → only 1 cycle delay

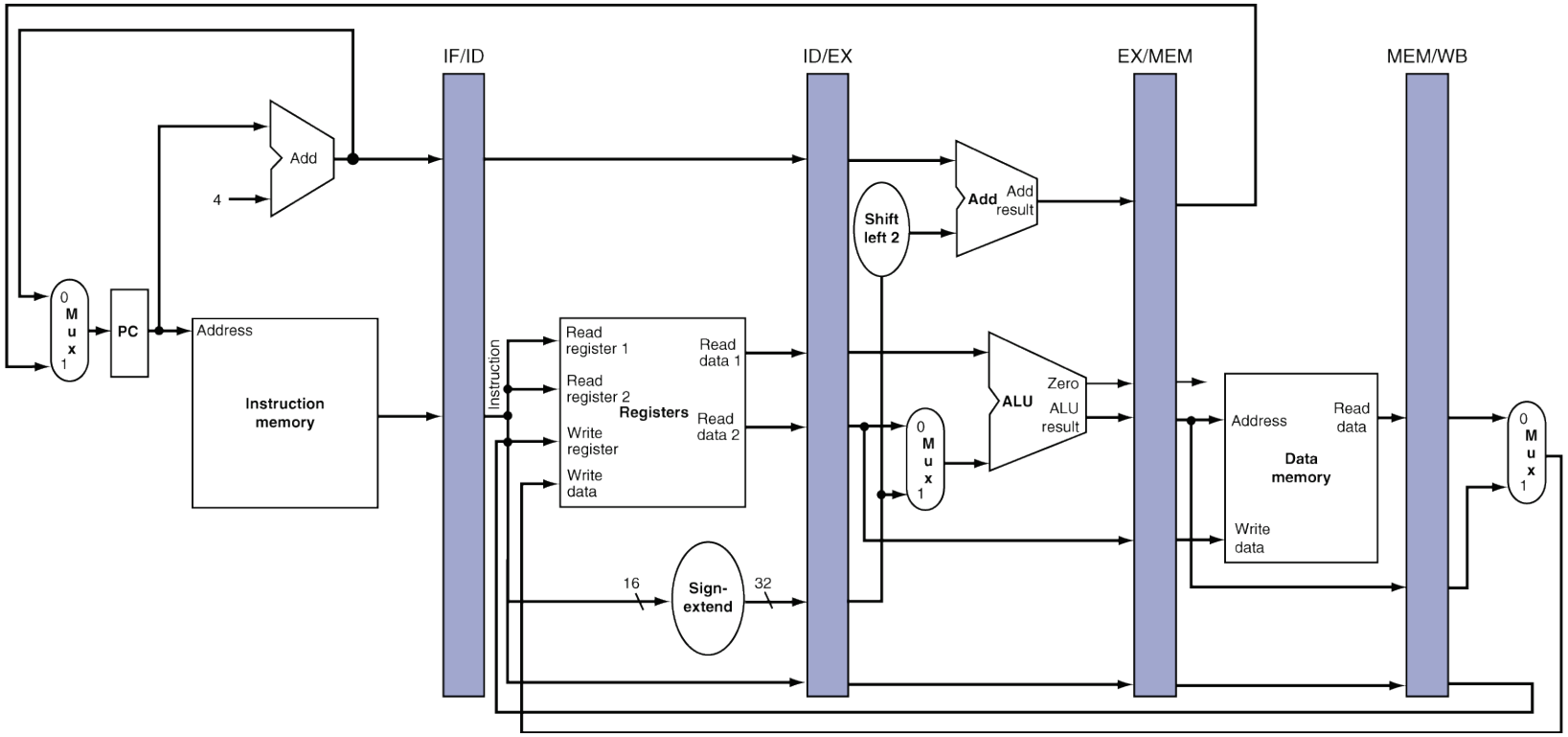
Review: Pipeline diagram



- Pipeline diagram shows execution of multiple instructions
  - Instructions listed vertically
  - Cycles shown horizontally
  - Each instruction divided into stages
  - Can see what instructions are in a particular stage at any cycle

Review: Pipeline registers

- Need registers between stages for info from previous cycles
- Register must be able to hold all needed info for given stage
  - For example, IF/ID must be 64 bits—32 bits for instruction, 32 bits for PC+4
- May need to propagate info through multiple stages for later use
  - For example, destination reg. number determined in ID, but not used until WB



Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Four Branch Hazard Alternatives

- #1: Stall until branch direction is clear
- #2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - "Squash" instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% MIPS branches not taken on average
  - PC+4 already calculated, so use it to get next instruction
- #3: Predict Branch Taken
  - 53% MIPS branches taken on average
  - But haven't calculated branch target address in MIPS
    - MIPS still incurs 1 cycle branch penalty
    - Other machines: branch target known before outcome

Four Branch Hazard Alternatives

#4: Delayed Branch

- Define branch to take place AFTER a following instruction

```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Downside: deeper pipelines/multiple issue → more branch delay
  - Dynamic approaches now more common
    - More expensive, but also more flexible
    - Will revisit in discussion of ILP

Instruction Level Parallelism

- Instruction-Level Parallelism (ILP): the ability to overlap instruction execution
- 2 approaches to exploit ILP & improve performance:
  1. Use hardware to dynamically find parallelism while running program
  2. Use software to find parallelism statically at compile-time

Finding ILP

- Basic Block (BB) ILP is quite small
  - BB: a code sequence with 1 entry and 1 exit point
  - average dynamic branch frequency 15% to 25%
    => 4 to 7 instructions execute between branches
  - Instructions in BB likely to depend on each other
- Must exploit ILP across multiple BB
- Simplest: loop-level parallelism
  - Parallelism across iterations of a loop:

```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

Loop-Level Parallelism

- Exploit loop-level parallelism by "unrolling loop" either

1. dynamically via branch prediction or
2. static via loop unrolling by compiler

- Determining instruction dependence is critical to Loop Level Parallelism
- If 2 instructions are
  - parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls
  - dependent, they are not parallel and must be executed in order, although they may often be partially overlapped
    - A stall caused by a dependence is a hazard

Static solution to LLP: Loop unrolling

- Loop iterations are often independent, e.g.

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

- Can create multiple copies of loop, then schedule them to avoid stalls
  - Replicate loop body, renaming registers as you go
  - Reorder appropriately to eliminate stalls
  - Update entry/exit code
- Unrolling goals
  - Cover all stalls (without using too many registers)
  - # old iterations should be divisible by # times unrolled
    - Example: loop with 100 iterations can be unrolled 2, 4, 5 times; not 3

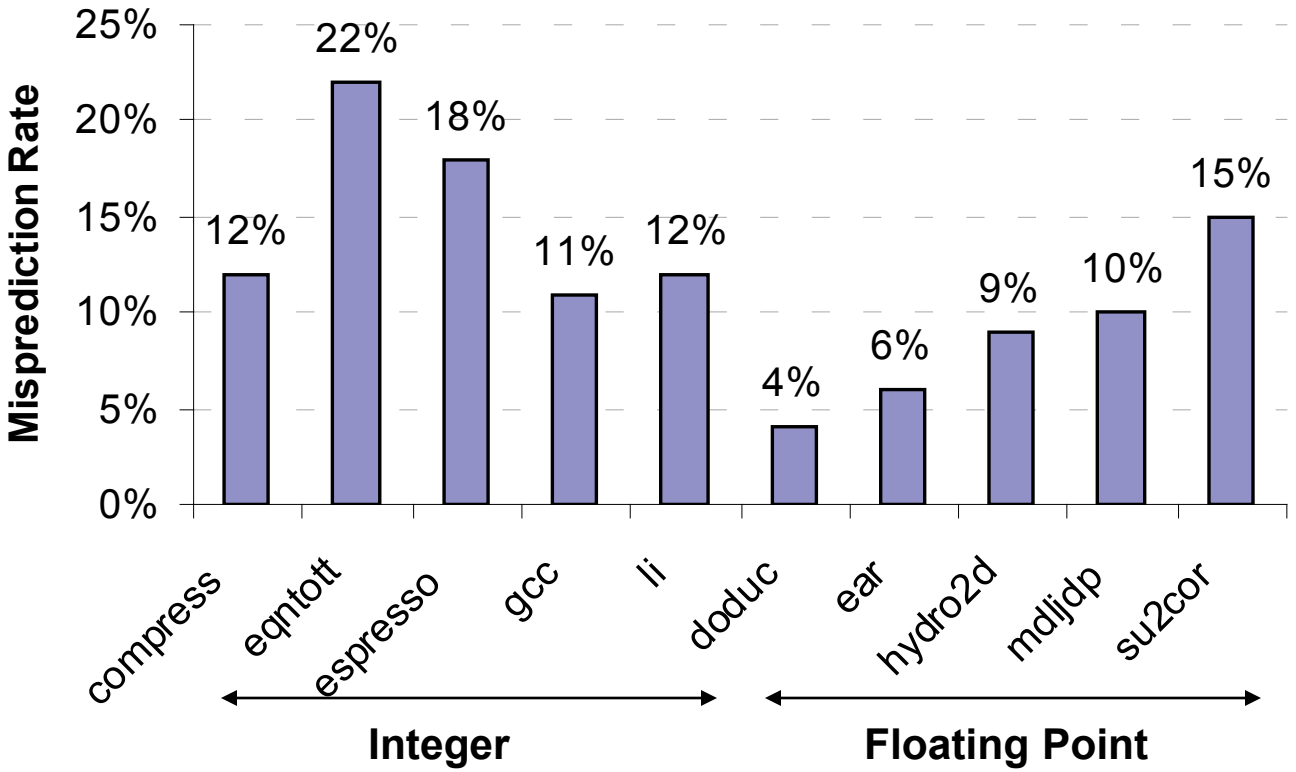
3 Limits to Loop Unrolling

- Decrease in amount of overhead amortized with each extra unrolling
  - Amdahl's Law
- Growth in code size
  - For larger loops, concern it increases the instruction cache miss rate
- Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
  - If not be possible to allocate all live values to registers, may lose some or all of its advantage
- Loop unrolling reduces branch impact on pipeline; another way is dynamic branch prediction

Static Branch Prediction

- Earlier lecture showed scheduling code around delayed branch
- To reorder code around branches, need to predict branch statically during compilation
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% SPEC

• More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run:



Dynamic Branch Prediction

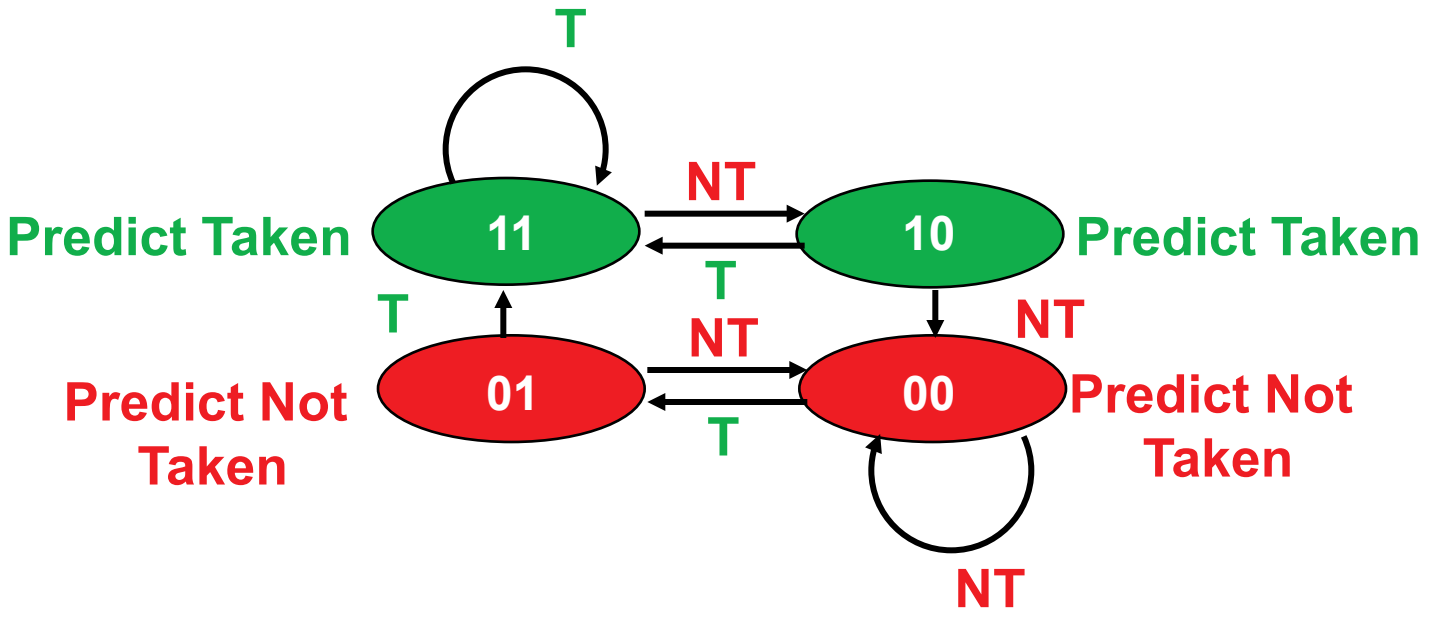
- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - There are a small number of important branches in programs which have dynamic behavior

Dynamic Branch Prediction

- Performance = *f*(accuracy, cost of misprediction)
- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts exit instead of looping

Dynamic Branch Prediction

- Solution: 2-bit scheme where change prediction only if get misprediction twice



- Red: “stop” (branch not taken)
- Green: “go” (branch taken)
- Adds hysteresis to decision making process

BHT example 1

- Simple loop with 1 branch, 4 iterations
- BHT state initially 00
- First iteration: predict NT, actually T
  - Update BHT entry: 00 → 01
- Second iteration: predict NT, actually T
  - Update BHT entry: 01 → 11
- Third iteration: predict T, actually T
  - No change to BHT entry
- Fourth iteration: predict T, actually NT
  - Update BHT entry: 11 → 10

BHT example 1

- Doesn't seem to be very helpful
  - 4 instances of branch executed, 3 mispredictions
- What if we return to the loop later?
  - Initial BHT entry state is 10
  - First iteration: predict T, actually T
    - Update BHT entry to 11
  - Second iteration: predict T, actually T
  - Third iteration: predict T, actually T
  - Fourth iteration: predict T, actually NT
    - Update BHT entry

  → 4 instances, only 1 misprediction

BHT example 2

- Given a nested loop:

```
Address
0         Loop1:  ...
8         Loop2:   ...
16                BNE R4, Loop2
20                ...
28                BEQ R7, Loop1
```

- Assume 4-entry BHT
- Questions:
  - How many bits to index?
    - And which ones?
  - What's initial prediction?
- Say inner loop has 8 iterations, outer loop has 4
  - How many mispredictions?

Line #	Prediction
0	00
1	00
2	00
3	00

BHT example 2 solution

- $4 = 2^2$ entries in BHT → 2 bits to index
- Use lowest order PC bits that actually change
  - All instructions 32 bits → lowest 2 bits always 0
  - Use next two bits
    - For address 16 = 0…0001 0000_2, line 0 of BHT
    - For address 28 = 0…0001 1100_2, line 3 of BHT

BHT example 2 solution (cont.)

- First iteration of outer loop
  - Reach inner loop for first time: BHT entry 0 = 00
    - First iteration: predict NT, actually T
      - Update BHT entry to 01
    - Second iteration: predict NT, actually T
      - Update BHT entry to 11
    - Third iteration: predict T, actually T
      - Can see that 4th-7th iterations will be exactly the same
    - Eighth iteration: predict T, actually NT
      - Update BHT entry to 10
  - Reach branch at end of outer loop: BHT entry 3 = 00
    - Predict NT, actually T
      - Update BHT entry to 01
- For this outer loop iteration
  - 5 correct predictions → Iterations 3-7 of inner loop
  - 4 mispredictions → Iterations 1, 2, & 8 of inner loop; iteration 1 of outer loop

BHT example 2 solution (cont.)

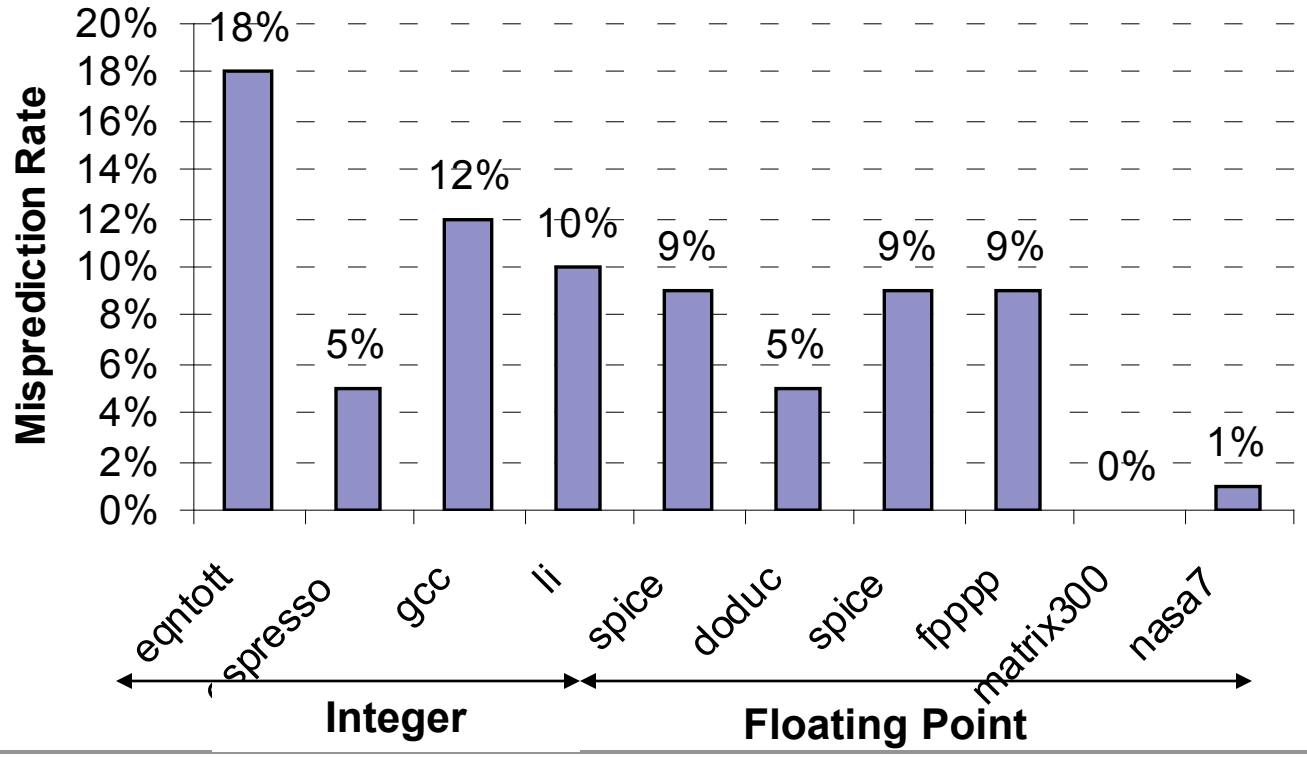
- Second iteration of outer loop
  - Reach inner loop: BHT entry 0 = 10
    - First iteration: predict T, actually T
      - Update BHT entry to 11
    - 2nd-7th iterations: predict T, actually T, no BHT entry transitions
    - Eighth iteration: predict T, actually NT
      - Update BHT entry to 10
  - Reach branch at end of outer loop: BHT entry 3 = 01
    - Predict NT, actually T
      - Update BHT entry to 11
- For this outer loop iteration
  - 7 correct predictions → Iterations 1-7 of inner loop
  - 2 mispredictions → Iteration 8 of inner loop; iteration 2 of outer loop

BHT example 2 solution (cont.)

- Third iteration of outer loop
  - Inner loop exactly the same
    - Predict iterations 1-7 correctly, 8 incorrectly
  - Correctly predict outer loop branch this time
    - BHT entry 3 = 11, so predict T and branch actually T
  - 8 correct predictions, 1 misprediction
- Fourth and final iteration of outer loop
  - Inner loop again the same
  - Outer loop branch mispredicted
    - Predict T, branch actually NT
  - 7 correct predictions, 2 mispredictions
- Overall
  - 5 + 7 + 8 + 7 = 27 correct predictions
  - 4 + 2 + 1 + 2 = 9 mispredictions
  - Misprediction rate = 9 / (9 + 27) = 9 / 36 = 25%

BHT Accuracy

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- 4096 entry table:



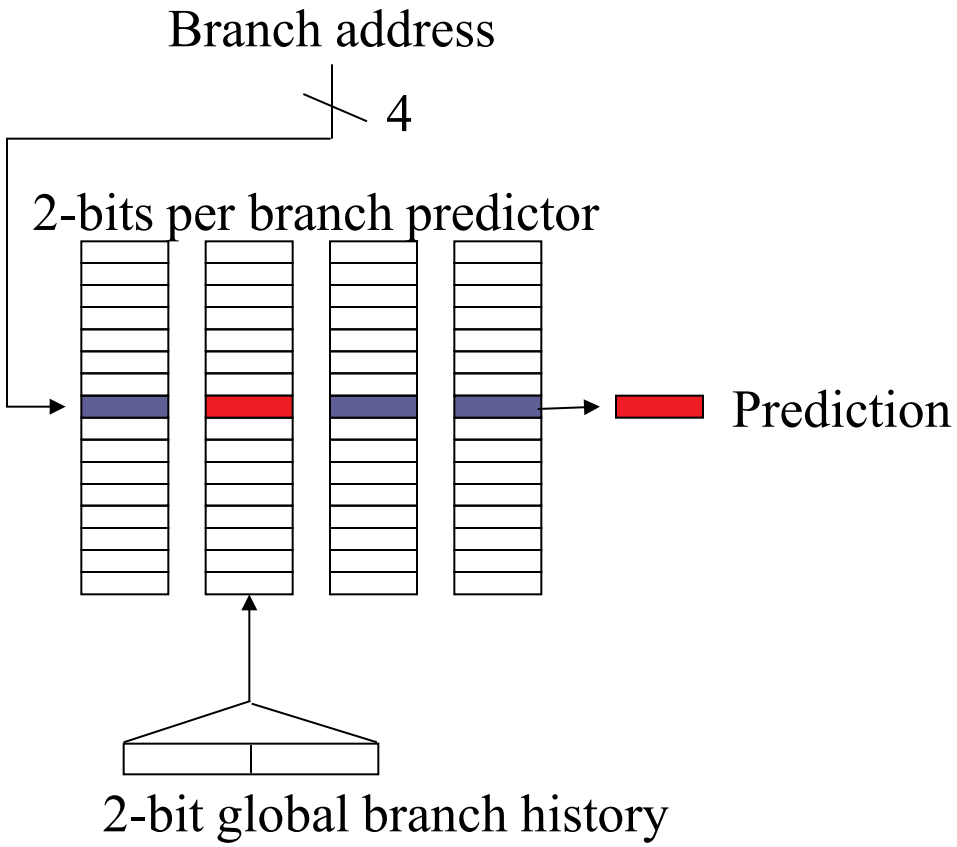
Correlated Branch Prediction

- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper n-bit branch history table
- In general, (m,n) predictor means record last m branches to select between 2^m history tables, each with n-bit counters
  - Thus, old 2-bit BHT is a (0,2) predictor
- Global Branch History: m-bit shift register keeping T/NT status of last m branches.
- Each entry in table has an n-bit predictor
  - Choose entry same way you do in a basic BHT (low-order address bits)

Correlating Branches

(2,2) predictor

- Behavior of recent branches selects between four predictions of next branch, updating just that prediction



Correlating example

- Look at one entry of a simple (2,2) predictor
- Assume
  - The program has been running for some time
  - Entry state is currently: (00, 10, 11, 01)
  - Global history is currently: 01
    - Last two branches were NT, T (T most recent)
- Say we have a branch accessing this entry
  - First 2 times, branch is taken
  - Next 2 times, branch is not taken
  - Final time, branch is taken

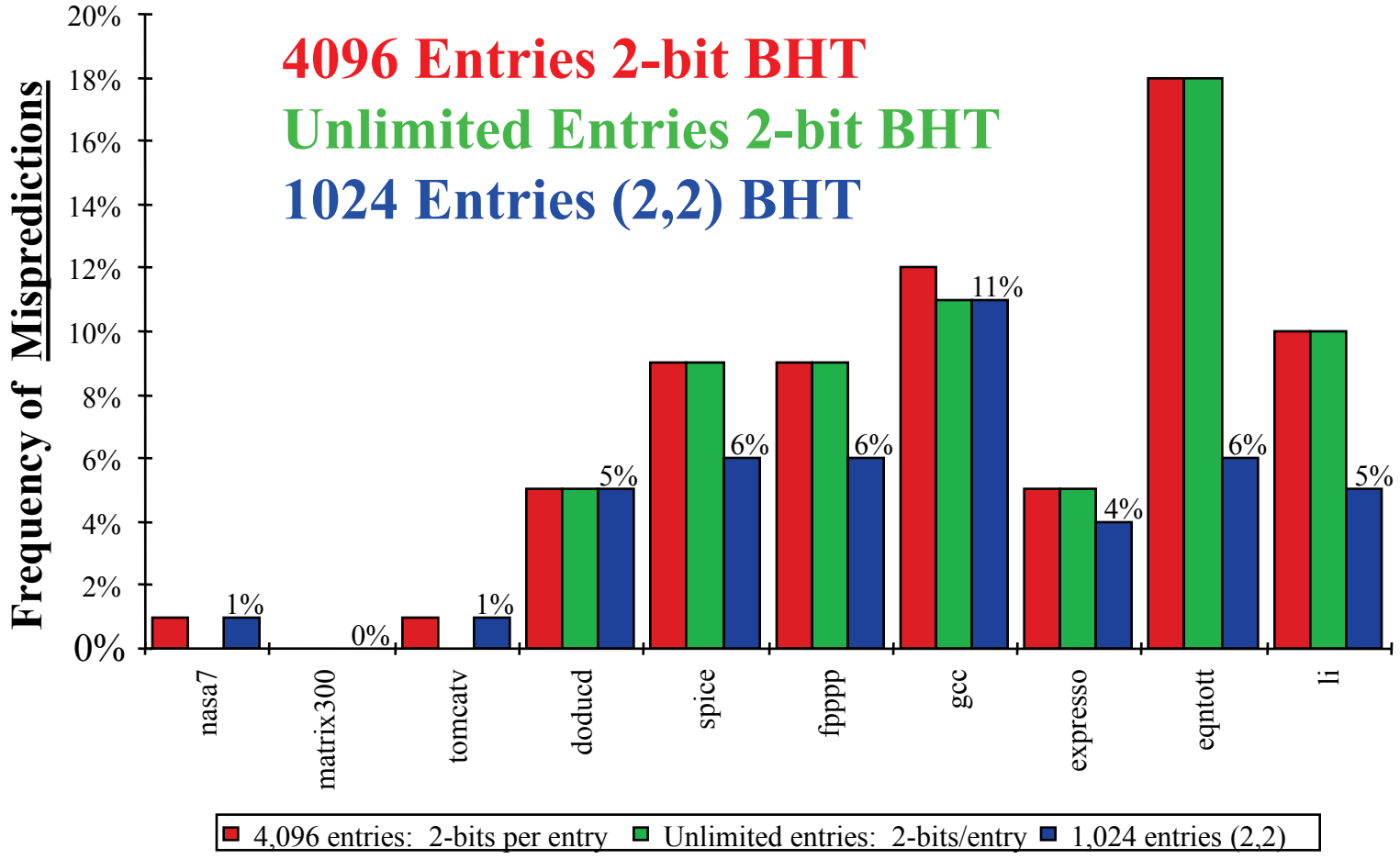
Correlating example (cont.)

- First access
  - Global history = 01 → entry[1] = 10
  - Predict T, actually T
    - Update entry[1] = 11
  - Update global history = 11
- Second access
  - Global history = 11 → entry[3] = 01
  - Predict NT, actually T
    - Update entry[3] = 11
  - Update global history = 11
    - Looks the same, but you are shifting in a 1

Correlating example (cont.)

- Third access
  - Global history = 11 → entry[3] = 11
  - Predict T, actually NT
    - Update entry[3] = 10
  - Update global history = 10
- Fourth access
  - Global history = 10 → entry[2] = 11
  - Predict T, actually NT
    - Update entry[2] = 10
  - Update global history = 00
- Fifth access
  - Global history = 00 → entry[0] = 00
  - Predict NT, actually T
    - Update entry[0] = 01
  - Update global history = 01

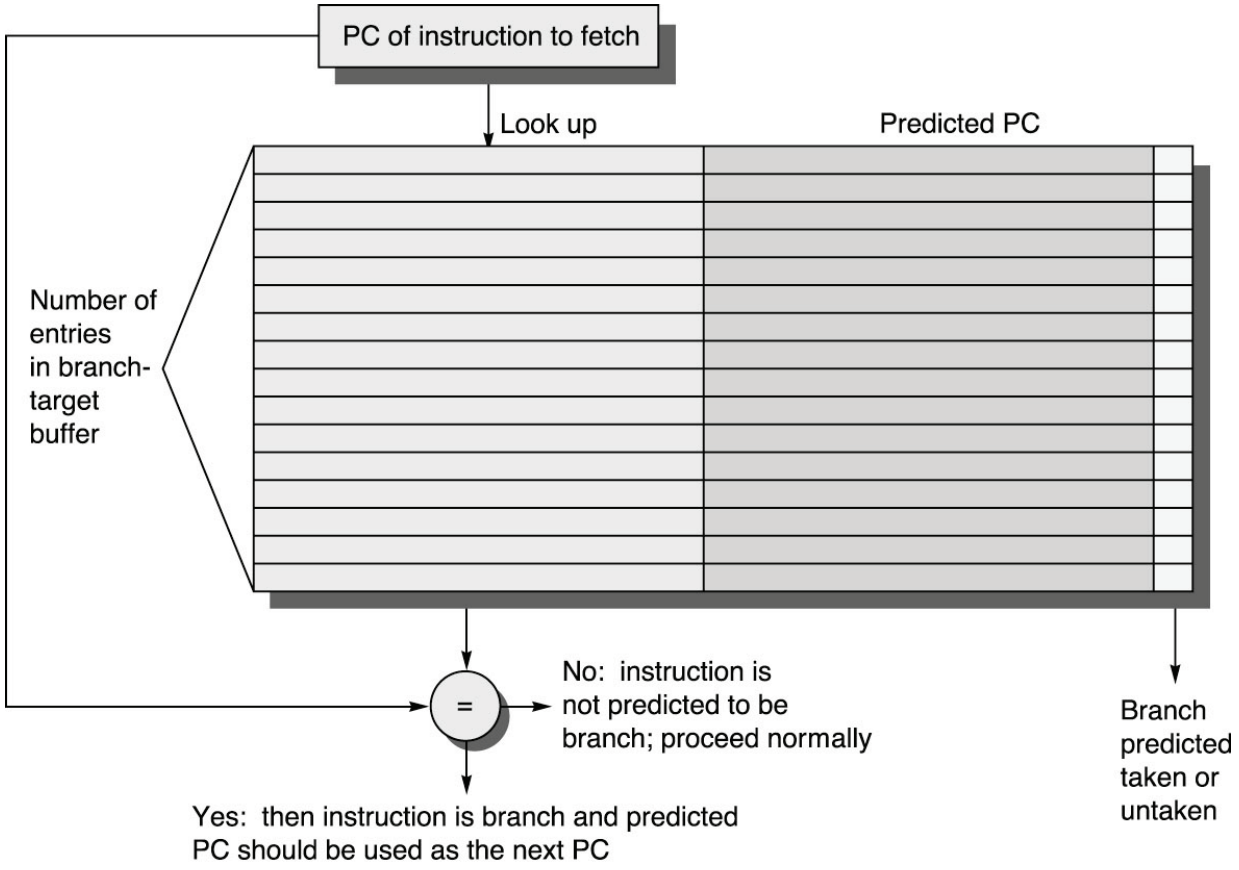
Accuracy of Different Schemes



Branch Target Buffers (BTB)

- Branch prediction logic: relatively fast
- Branch target calculation is slower
  - Must actually decode instruction
  - To remove stalls in speculative execution, need target more quickly
    - Store previously calculated targets in branch target buffer
- Send PC of branch to the BTB
  - Check if matching address exists (tag check, like cache)
- If match is found, corresponding Predicted PC is returned
- If the branch was predicted taken, instruction fetch continues at the returned predicted PC

Branch Target Buffers



Final notes

- Next time:
  - Instruction scheduling issues
    - Dynamic scheduling
    - Multiple issue
- Announcements/reminders
  - HW 2 due today
  - HW 3 to be posted; due 2/27