CS151B
Computer Systems Architecture

Week 7 Discussion

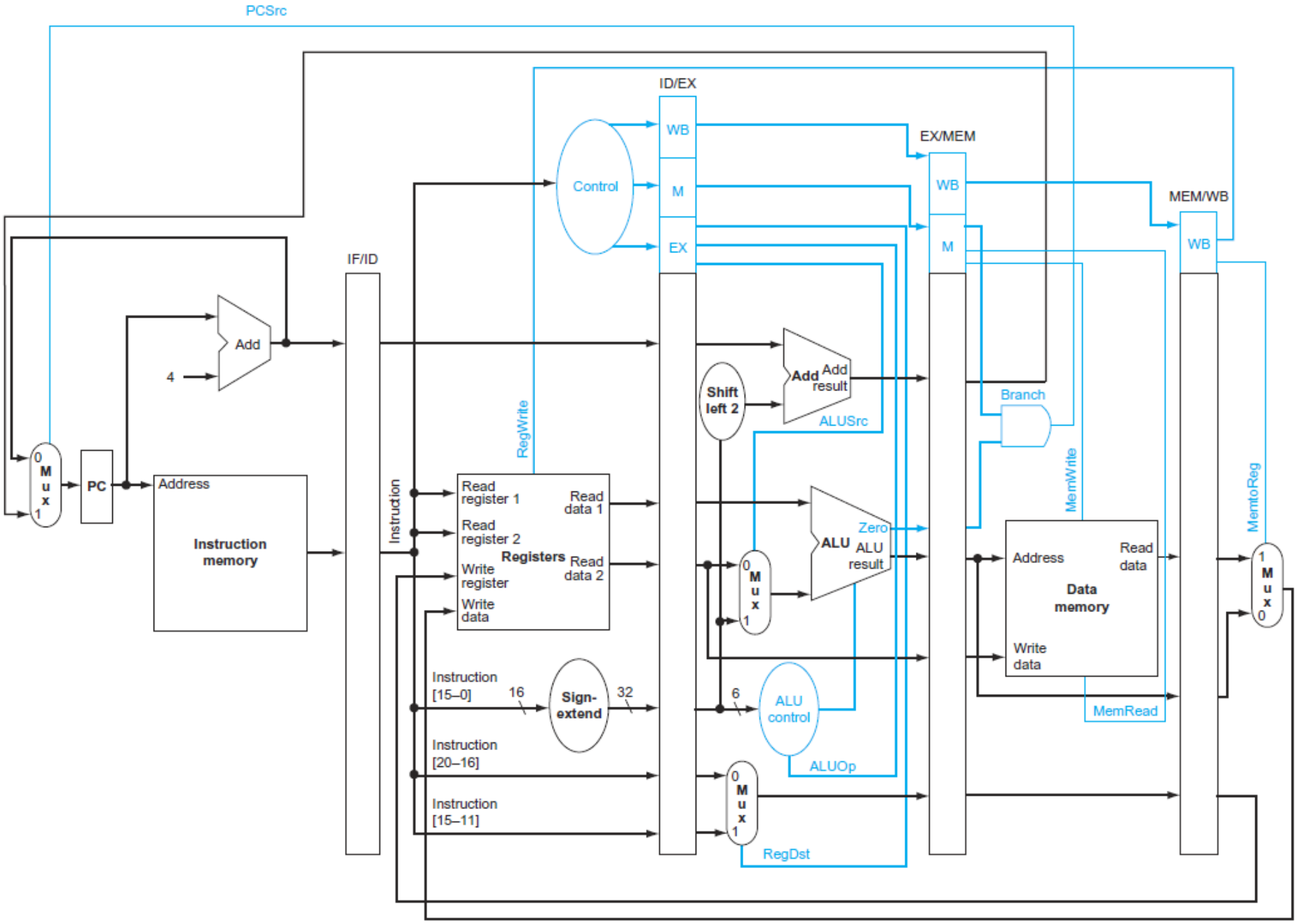
2/23/2018

Logistics

- HW5 due today
- HW6 due next Friday

Agenda

- Control hazards
  - Branch prediction


PCSrc
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
ALU
Zero
ALU result
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
16
Sign-extend
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst

Control Dependence

- Question: What should the fetch PC be in the next cycle?
- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next fetch PC?
- In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions

- Potential solutions if the instruction is a control-flow instruction
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (**branch prediction**)
  - Employ delayed branching (**branch delay slot**)
  - Do something else (**fine-grained multithreading**)
  - Eliminate control-flow instructions (**predicated execution**)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (**multipath execution**)

Delayed Branching

- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles

- Idea: **Delay the execution of a branch. N instructions (delay slots) that come after the branch are *always* executed regardless of branch direction.**

- Problem: **How do you find instructions to fill the delay slots?**
  - **Branch must be independent of delay slot instructions**

- Unconditional branch: Easier to find insts to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots -> difficult to fill the delay slot

Delayed Branching

- Advantages:
  - Keeps the pipeline full with useful instructions in a simple way assuming
    - Number of delay slots == number of instructions to keep the pipeline full before the branch resolves
    - All delay slots can be filled with useful instructions

- Disadvantages:
  - Not easy to fill the delay slots (even with a 2-stage pipeline)
    - Number of delay slots increases with pipeline depth, superscalar execution width
  - Ties ISA semantics to hardware implementation
    - SPARC, MIPS: 1 delay slot
    - What if pipeline implementation changes with the next design?

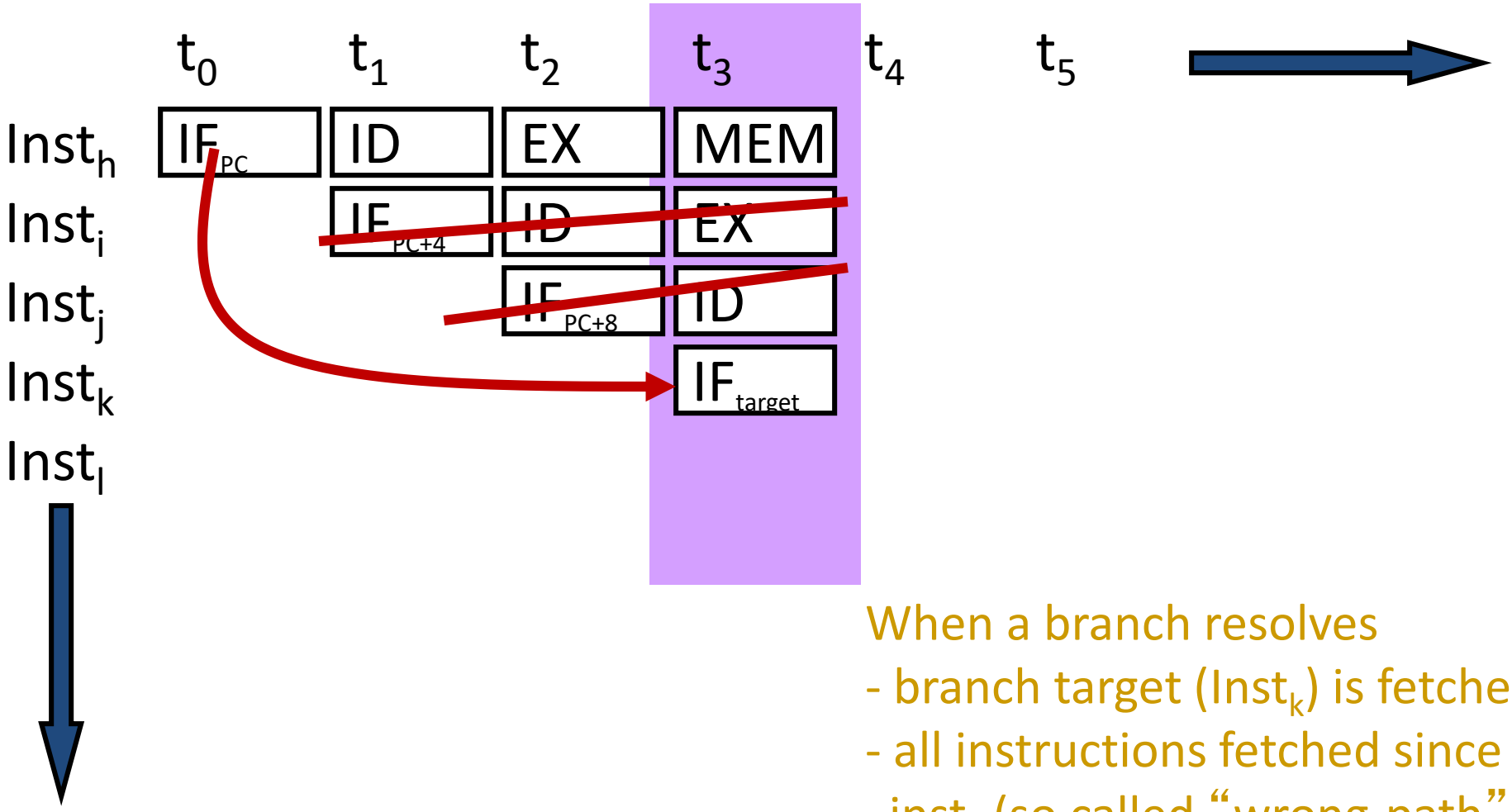
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions

- Potential solutions if the instruction is a control-flow instruction
  - Stall the pipeline until we know the next fetch address
  - Guess the next fetch address (**branch prediction**)
  - Employ delayed branching (**branch delay slot**)
  - Do something else (**fine-grained multithreading**)
  - Eliminate control-flow instructions (**predicated execution**)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (**multipath execution**)

Branch Prediction

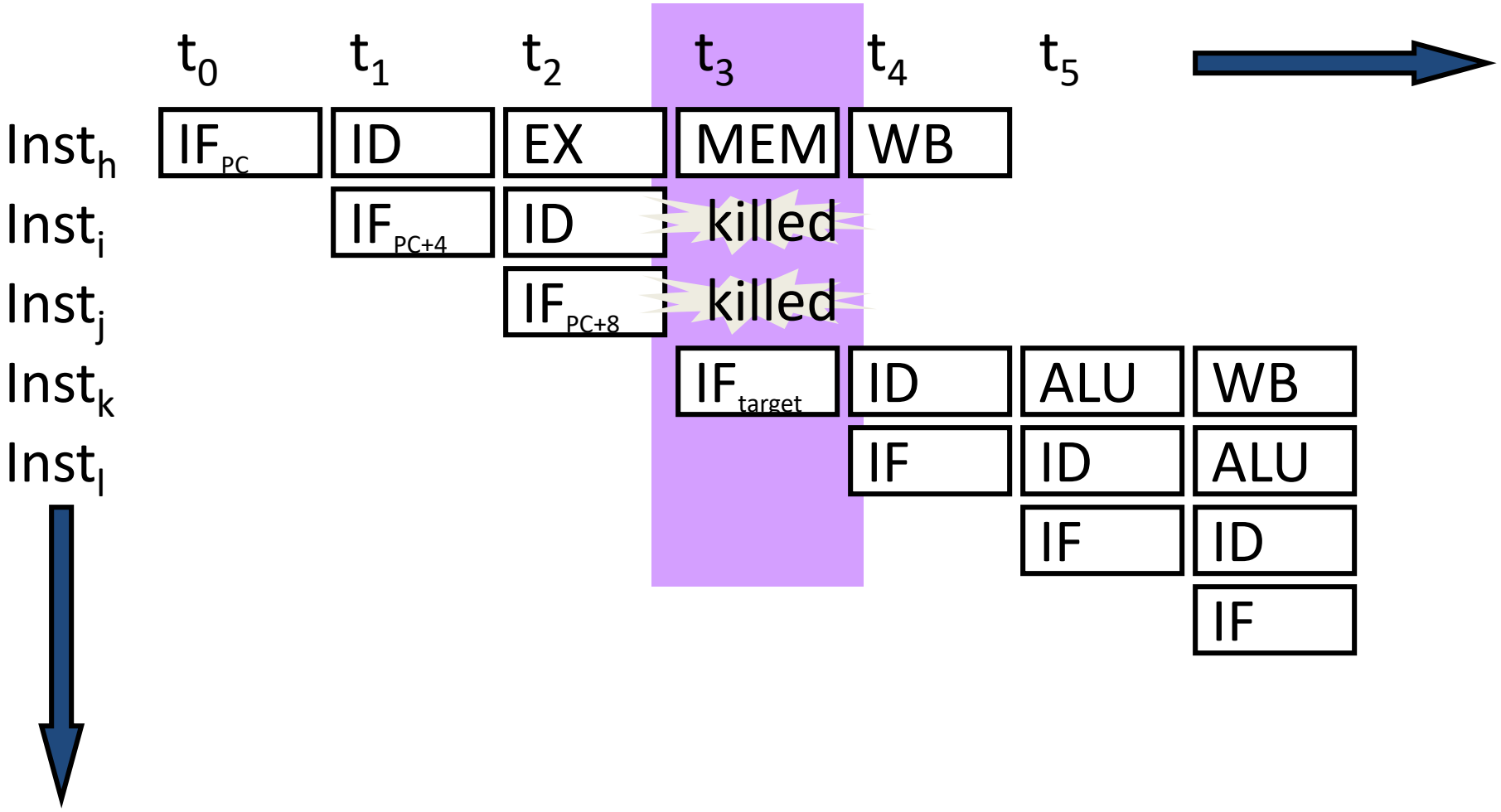
Branch Prediction: Always PC+4 (EX Resolve)



When a branch resolves
- branch target ($Inst_k$) is fetched
- all instructions fetched since $inst_h$ (so called "wrong-path" instructions) must be flushed

$Inst_h$ is a branch

Pipeline Flush on a Misprediction (EX Resolve)



$Inst_h$ is a branch

Performance Analysis

- correct guess ⇒ no penalty ~86% of the time
- incorrect guess ⇒ 2 bubbles (EX resolve)
- Assume
  - no data dependency related stalls
  - 20% control flow instructions
  - 70% of control flow instructions are taken
  - CPI = [1 + (0.20*0.7) * 2] =
    = [1 + 0.14 * 2] = 1.28

probability of a wrong guess

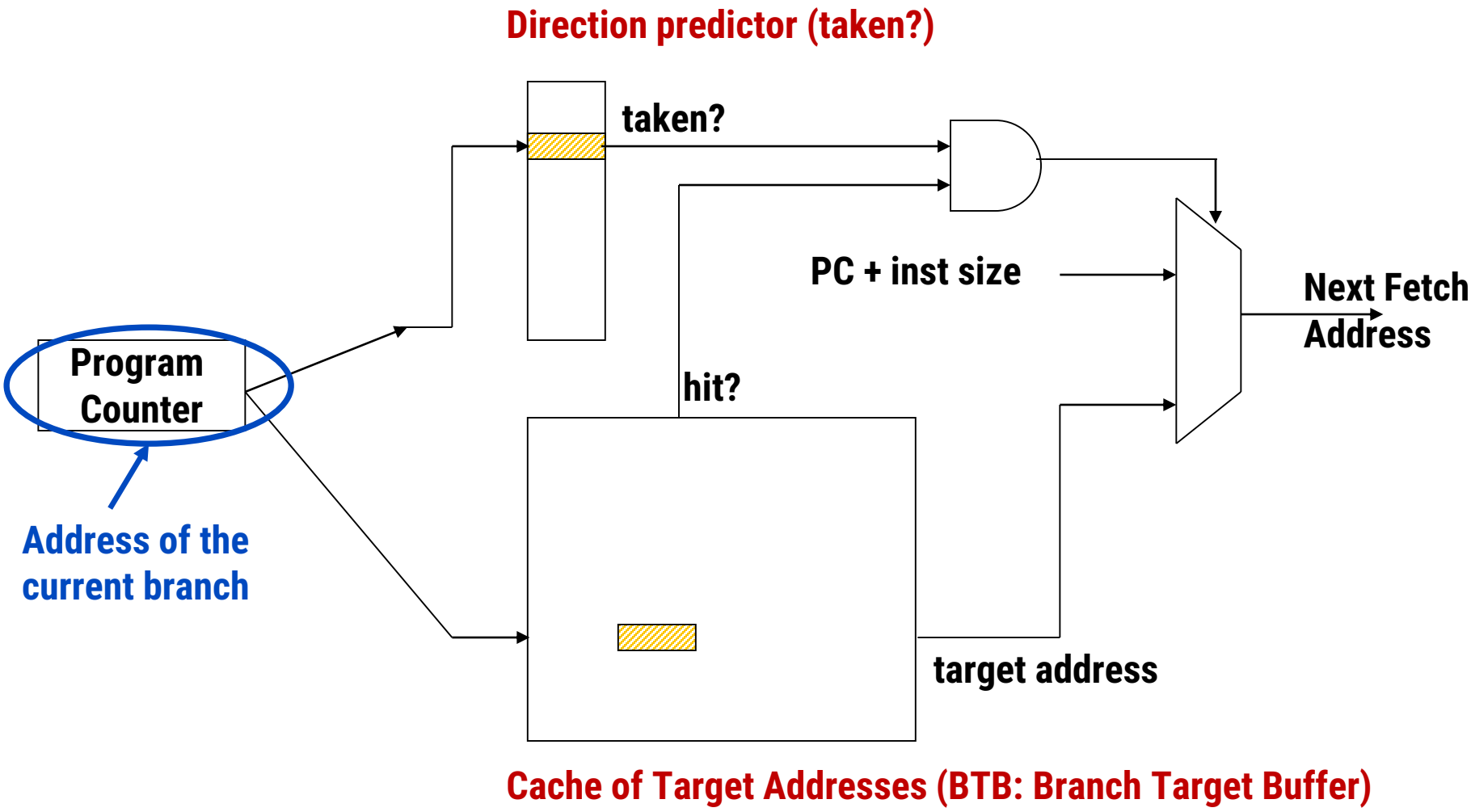
penalty for a wrong guess

Can we reduce either of the two penalty terms?

Branch Prediction

- Idea: **Predict the next fetch address (to be used in the next cycle)**
- Requires three things to be predicted at fetch stage:
  - **Whether the fetched instruction is a branch**
  - **(Conditional) branch direction**
  - **Branch target address**
- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - Idea: **Store the target address from previous instance and access it with the PC**
  - Called **Branch Target Buffer (BTB)**

Fetch Stage with BTB and Direction Prediction



Three Things to Be Predicted

- Requires three things to be predicted at fetch stage:
  - 1. Whether the fetched instruction is a branch
  - 2. (Conditional) branch direction
  - 3. Branch target address (if taken)

- Third (3.) can be accomplished using a BTB
  - Remember target address computed last time branch was executed

- First (1.) can be accomplished using a BTB
  - If BTB provides a target address for the program counter, then it must be a branch

- Second (2.): How do we predict the direction?

More Sophisticated Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

- Run time (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid

Static Branch Prediction (I)

- **Always not-taken**
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40% (for conditional branches)
  - Remember: Compiler can layout code such that the likely path is the "not-taken" path → more effective prediction
- **Always taken**
  - No direction prediction
  - Better accuracy: ~60-70% (for conditional branches)
    - 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

Dynamic Branch Prediction

- Idea: **Predict branches based on dynamic information** (collected at run-time)

- Advantages
  - Prediction based on history of the execution of branches
  - It can adapt to dynamic changes in branch behavior
  - No need for static profiling: input set representativeness problem goes away

- Disadvantages
  - More complex (requires additional hardware)

Last Time Predictor

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

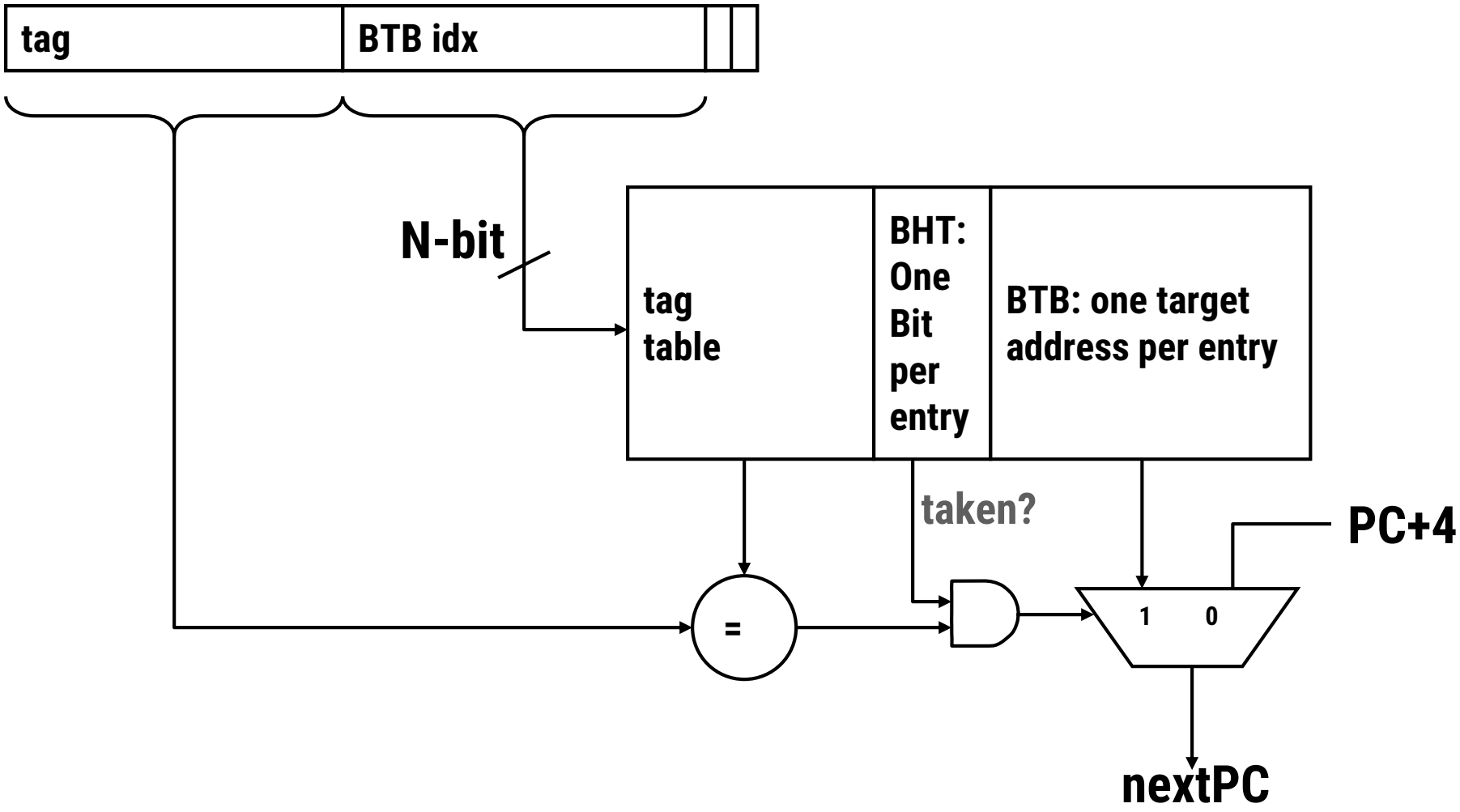
- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N - 2) / N

+ Loop branches for loops with large N (number of iterations)

-- Loop branches for loops with small N (number of iterations)

TNTNTNTNTNTNTNTNTNTN → 0% accuracy

Implementing the Last-Time Predictor



The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch

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

- Smith, "A Study of Branch Prediction Strategies," ISCA 1981.

Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome

- Accuracy for a loop with N iterations = (N - 1) / N
  TNTNTNTNTNTNTNTNTNTNTN → 50% accuracy
  (assuming counter initialized to weakly taken)

+ Better prediction accuracy
-- More hardware cost

Is This Good Enough?

- ~85-90% accuracy for many programs with 2-bit counter based prediction
- Is this good enough?
- How big is the branch problem?

Rethinking the The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions
- Problem: **Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor**
  - N cycles: (minimum) branch resolution latency
- If we are fetching W instructions per cycle (i.e., if the pipeline is W wide)
  - **A branch misprediction leads to N x W wasted instruction slots**

Importance of The Branch Problem

- Assume N = 20 (20 pipe stages), W = 5 (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

- How long does it take to fetch 500 instructions?
  - 100% accuracy
    - **100 cycles** (all instructions fetched on the correct path)
    - No wasted work
  - 99% accuracy
    - 100 (correct path) + 20 (wrong path) = **120 cycles**
    - **20% extra instructions fetched**
  - 98% accuracy
    - 100 (correct path) + 20 * 2 (wrong path) = **140 cycles**
    - **40% extra instructions fetched**
  - 95% accuracy
    - 100 (correct path) + 20 * 5 (wrong path) = **200 cycles**
    - **100% extra instructions fetched**

Slides Credit

- Onur Mutlu