Branch Prediction: speculation in HW

Seehwan Yoo

Dankook University

Dept. of Mobile Systems

DKU DANKOOK UNIVERSITY

Review: Jump Types

Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Different branch types can be handled differently

Review: How to Handle Control Dependences

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

- 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)

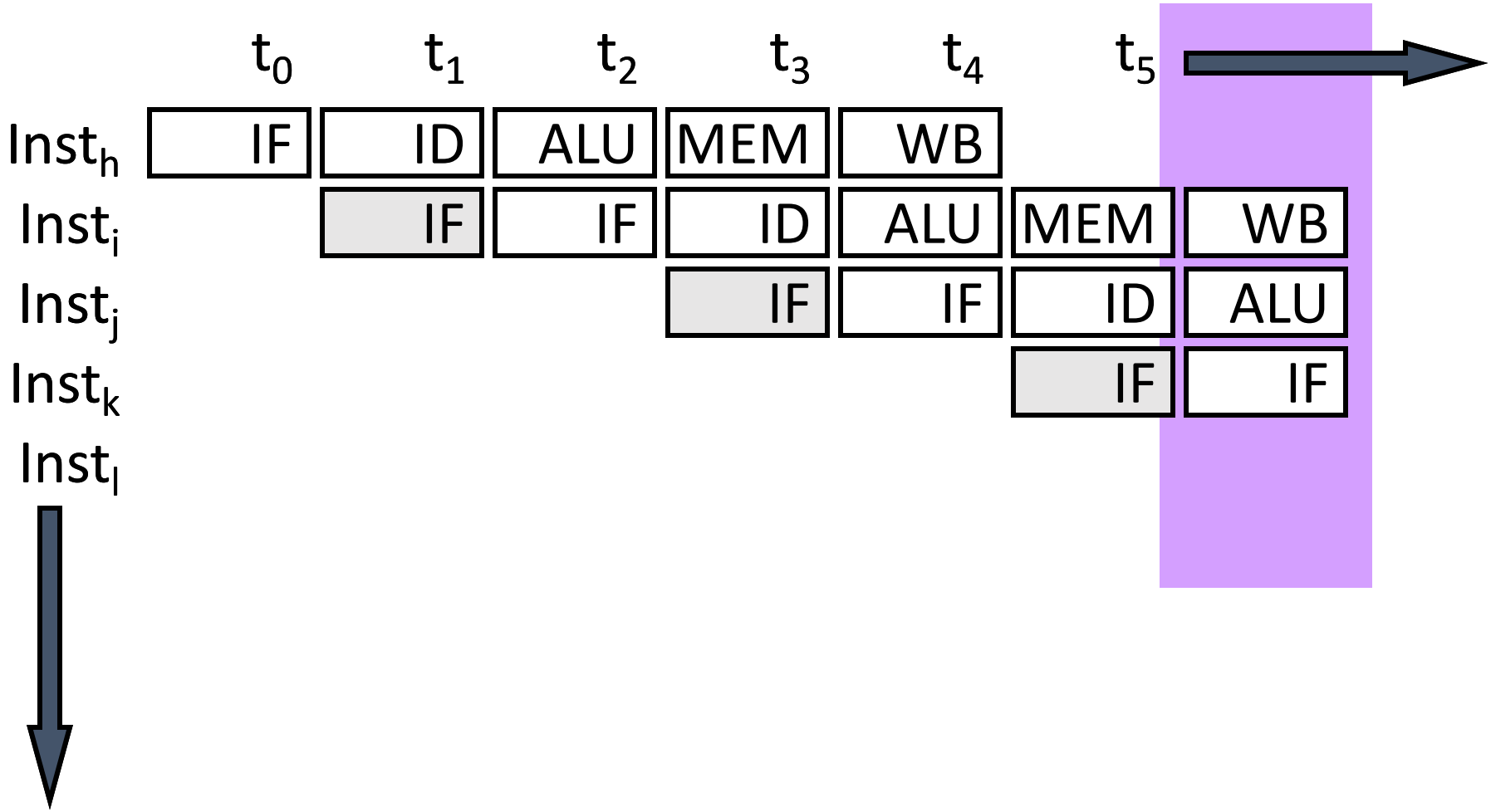
Difficulties in control dependency

- Fetch address is not known until the end of execution stage
- What if 20 stages pipeline?
  - Execution stage completes after 10 cycles
  - Where to fetch instructions for the 10 cycles?
  - Stalling 10 cycles, seems too many
- What if branch occurs frequently?
  - 15~20% branch instructions?

Review: Questions regarding the fetch after branch

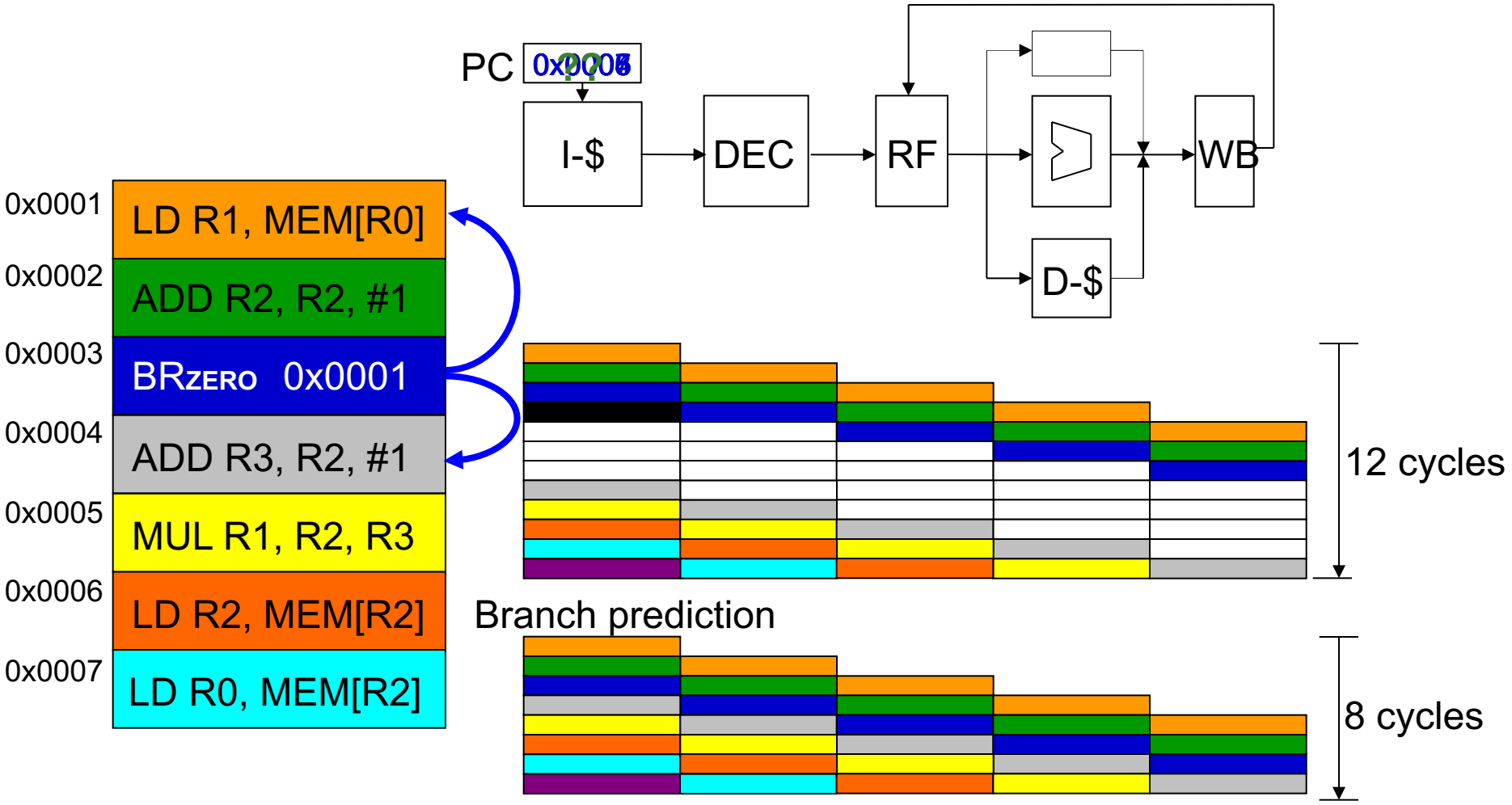
- Fetching instruction after branch
  - Filling in the pipeline
- Q1) Is the instruction branch? (is branch or not?)
- Q2) Will the branch be taken? (will jump or not?)
- Q3) Where is the target if it is taken? (target address)

Stall Fetch Until Next PC is Available: Good Idea?

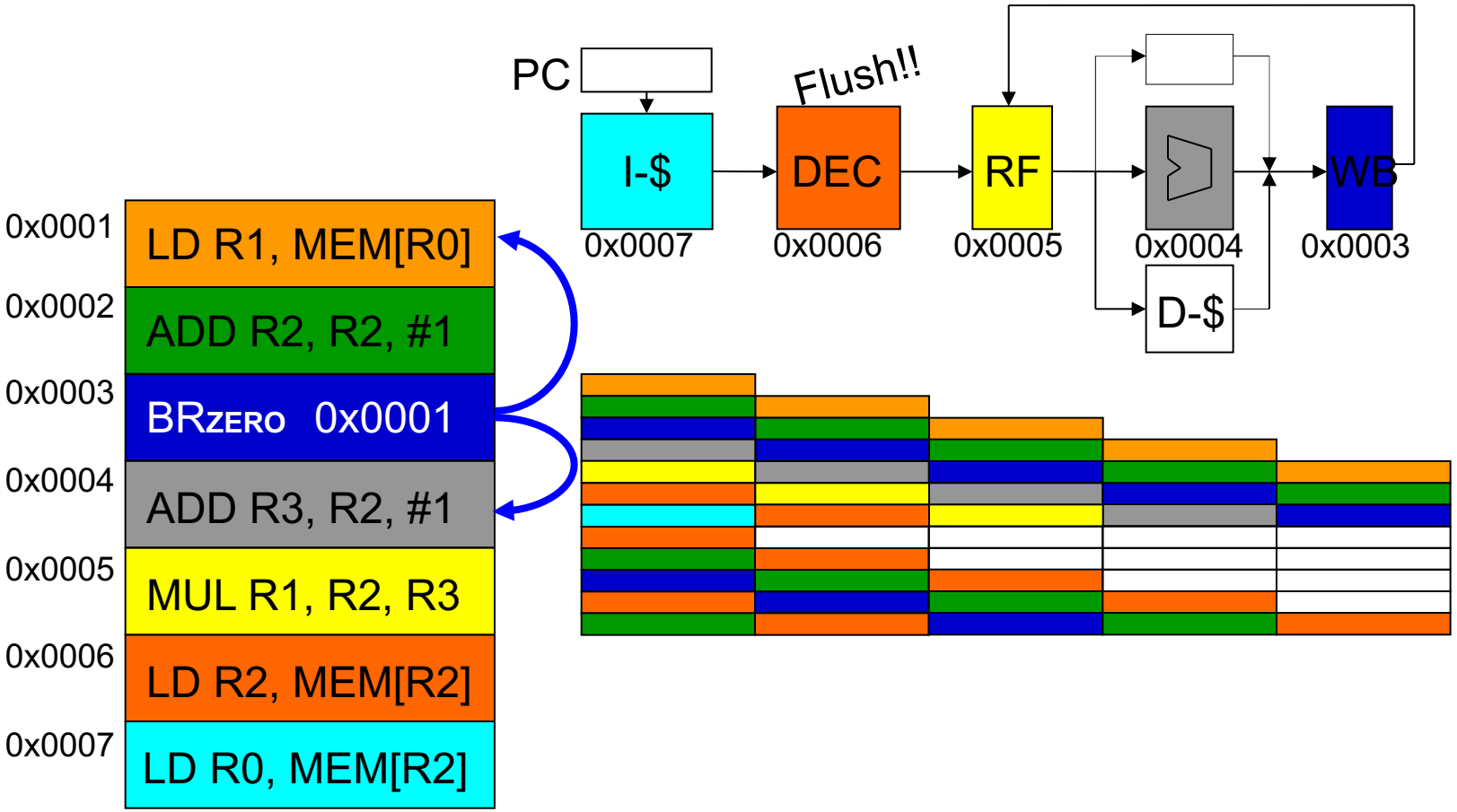


This is the case with non-control-flow and unconditional br instructions!

Branch Prediction: Guess the Next Instruction to Fetch

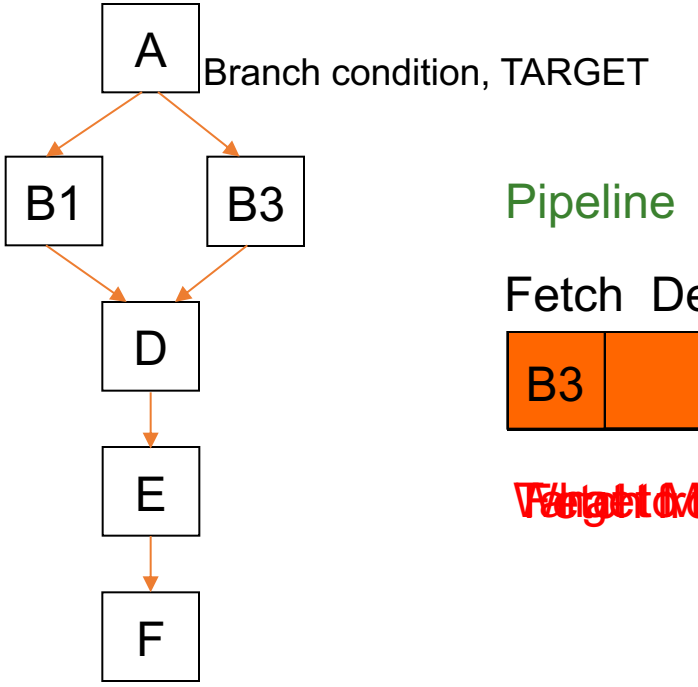


Misprediction Penalty

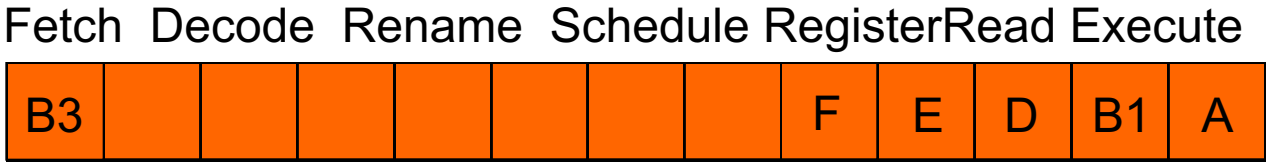


Branch Prediction

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch



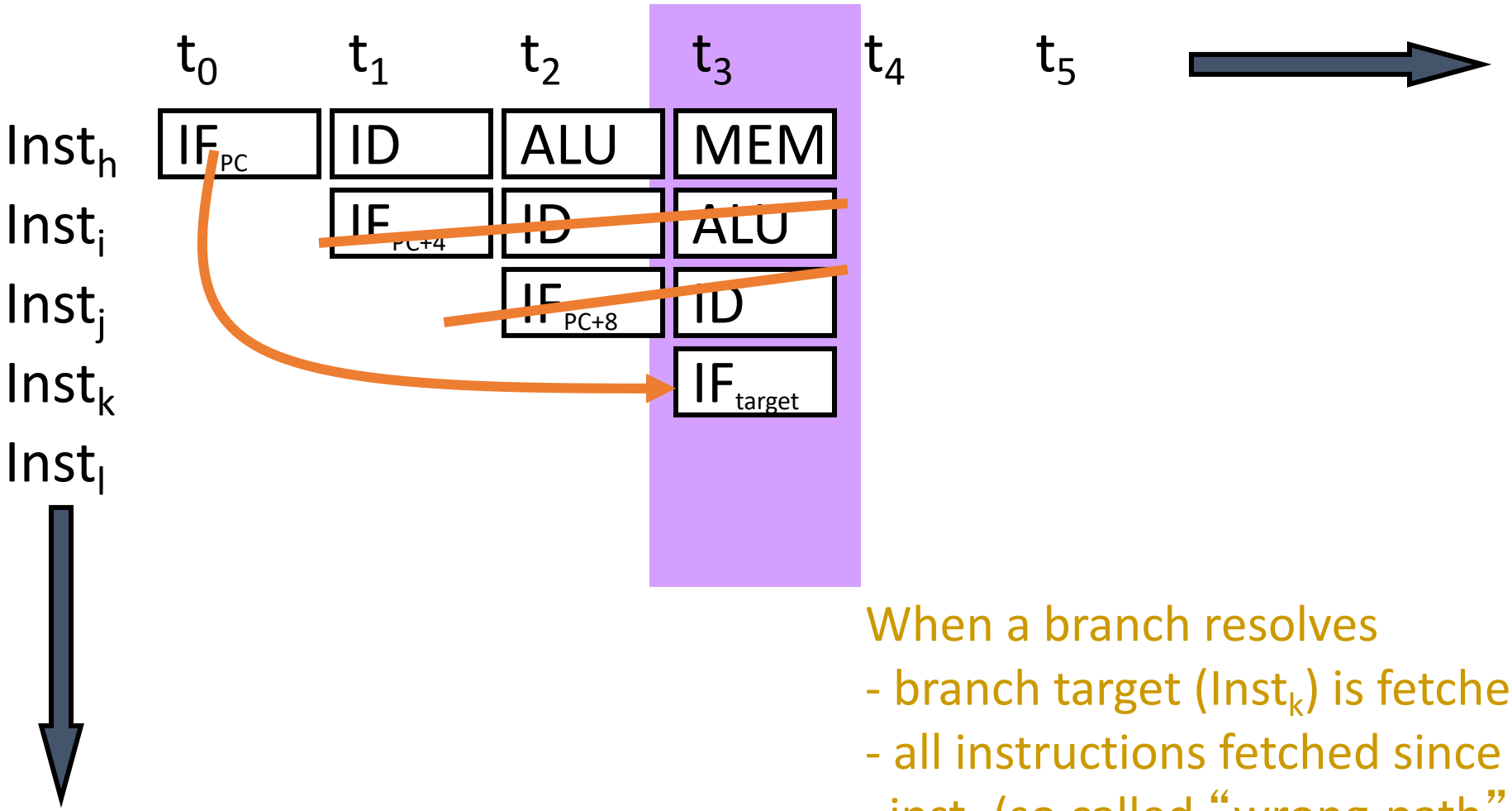
Pipeline



Target Misprediction Detected! Flush the Pipeline

DANKOOK UNIVERSITY

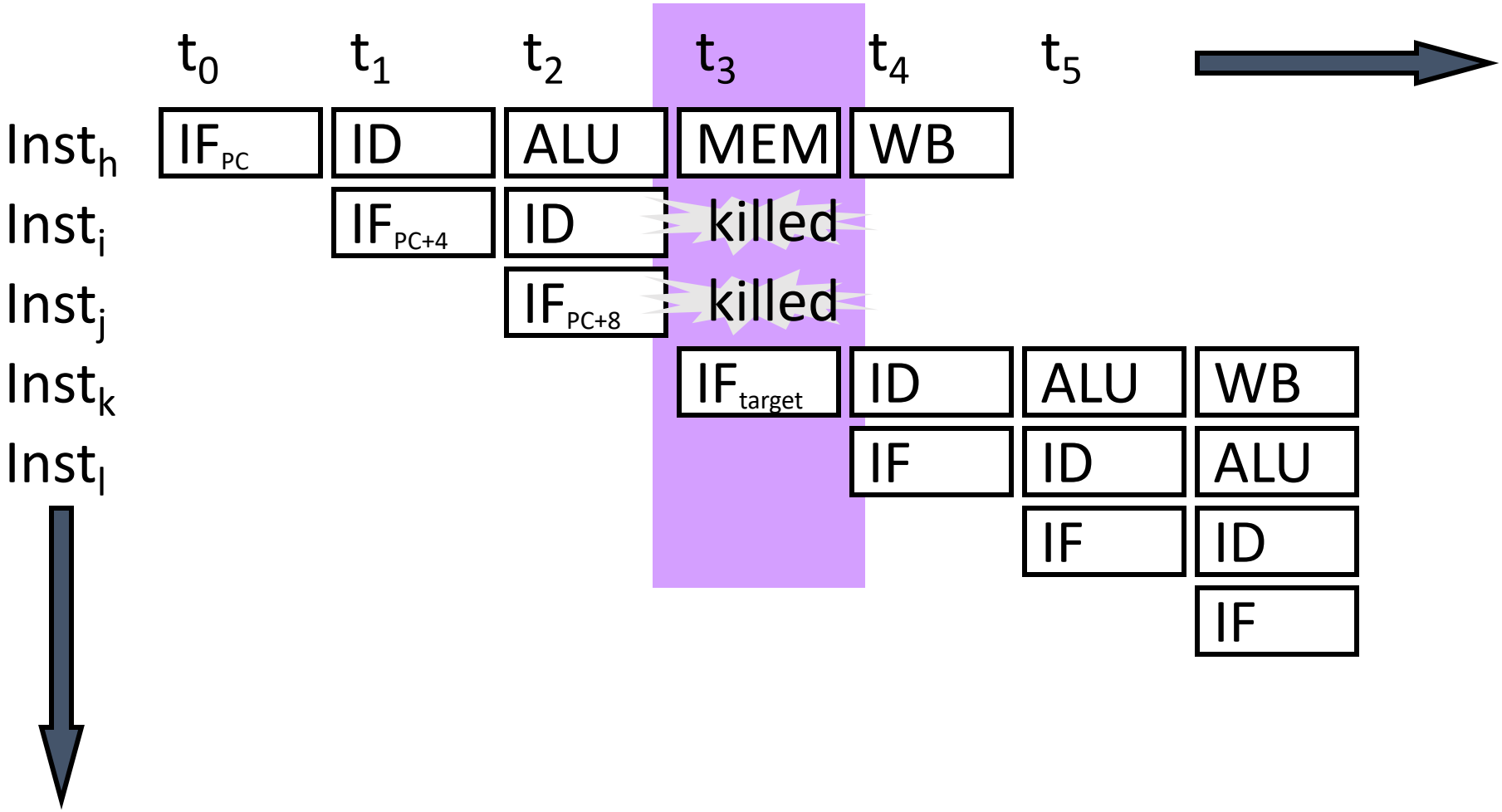
Simplest Branch Prediction: Always PC+4



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



Insth is a branch

Performance Analysis

- correct guess ⇒ no penalty ~86% of the time
- incorrect guess ⇒ 2 bubbles
- Assume
  - no data hazards
  - 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?

Static Branch Prediction (I)

- 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

Static Branch Prediction (II)

- Profile-based
  - Idea: Compiler determines likely direction for each branch using profile run. Encodes that direction as a hint bit in the branch instruction format.

+ Per branch prediction (more accurate than schemes in previous sli de) → accurate if profile is representative!

-- Requires hint bits in the branch instruction format

-- Accuracy depends on dynamic branch behavior:

TTTTTTTTTTNNNNNNNNNN → 50% accuracy TNTNTNTNT NTNTNTNTNTN → 50% accuracy

-- Accuracy depends on the representativeness of profile input set

Static Branch Prediction (III)

- Program-based (or, program analysis based)
  - Idea: Use heuristics based on program analysis to determine statically-predicted direction
  - Opcode heuristic: Predict BLEZ as NT (negative integers used as error values in many programs)
  - Loop heuristic: Predict a branch guarding a loop execution as taken (i.e., execute the loop)
  - Pointer and FP comparisons: Predict not equal

+ Does not require profiling

-- Heuristics might be not representative or good

-- Requires compiler analysis and ISA support

- Ball and Larus, "Branch prediction for free," PLDI 1993.
  - 20% misprediction rate

Static Branch Prediction (III)

- Programmer-based
  - Idea: Programmer provides the statically-predicted direction
  - Via pragmas in the programming language that qualify a branch as li kely-taken versus likely-not-taken

+ Does not require profiling or program analysis

+ Programmer may know some branches and their program better than othe r analysis techniques

-- Requires programming language, compiler, ISA support

-- Burdens the programmer?

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

Dynamic Branch Prediction

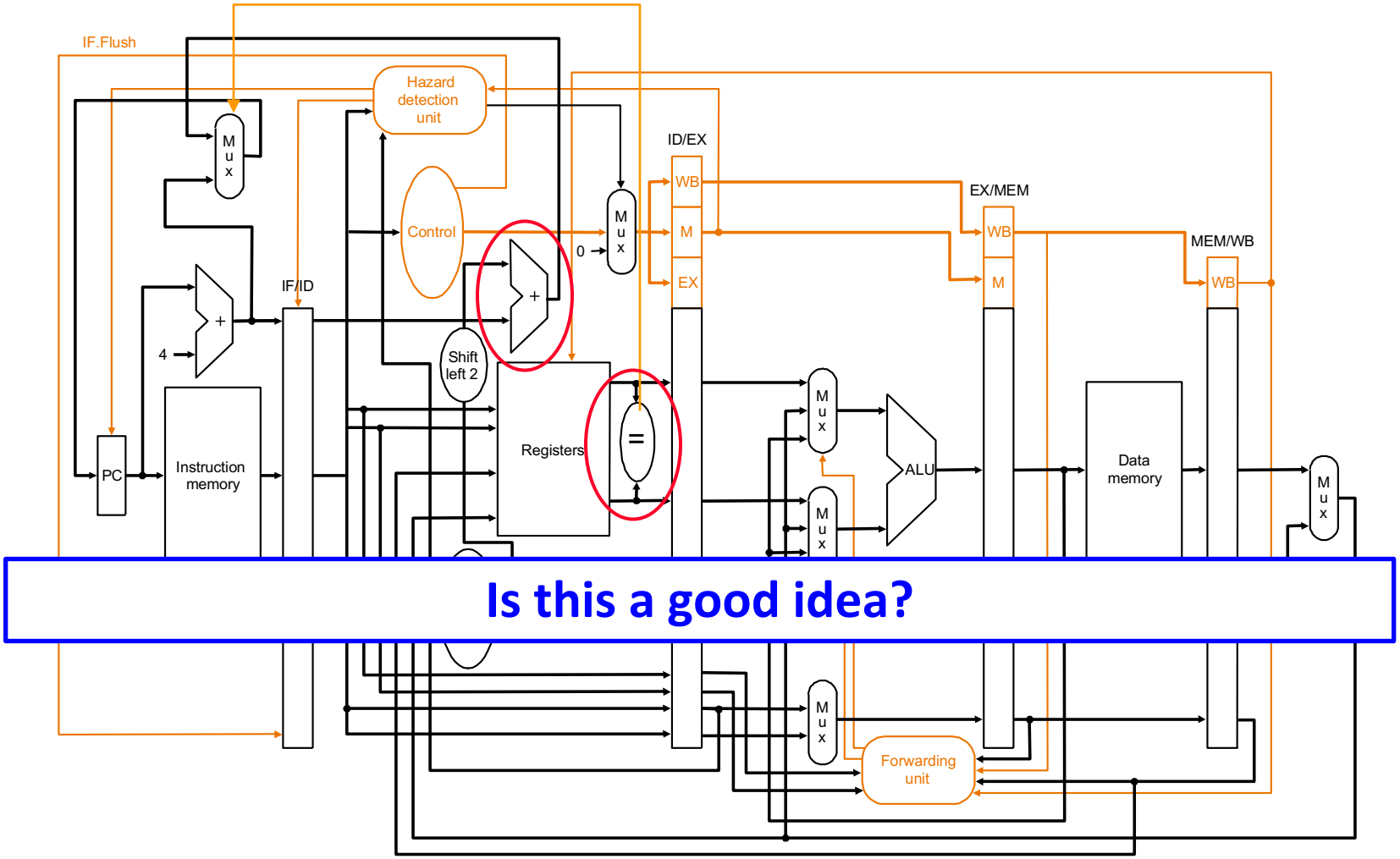
- Idea: Predict the next fetch address (to be used in the next cycle)

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

- 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) or Branch Target Address Cache(BTAC)

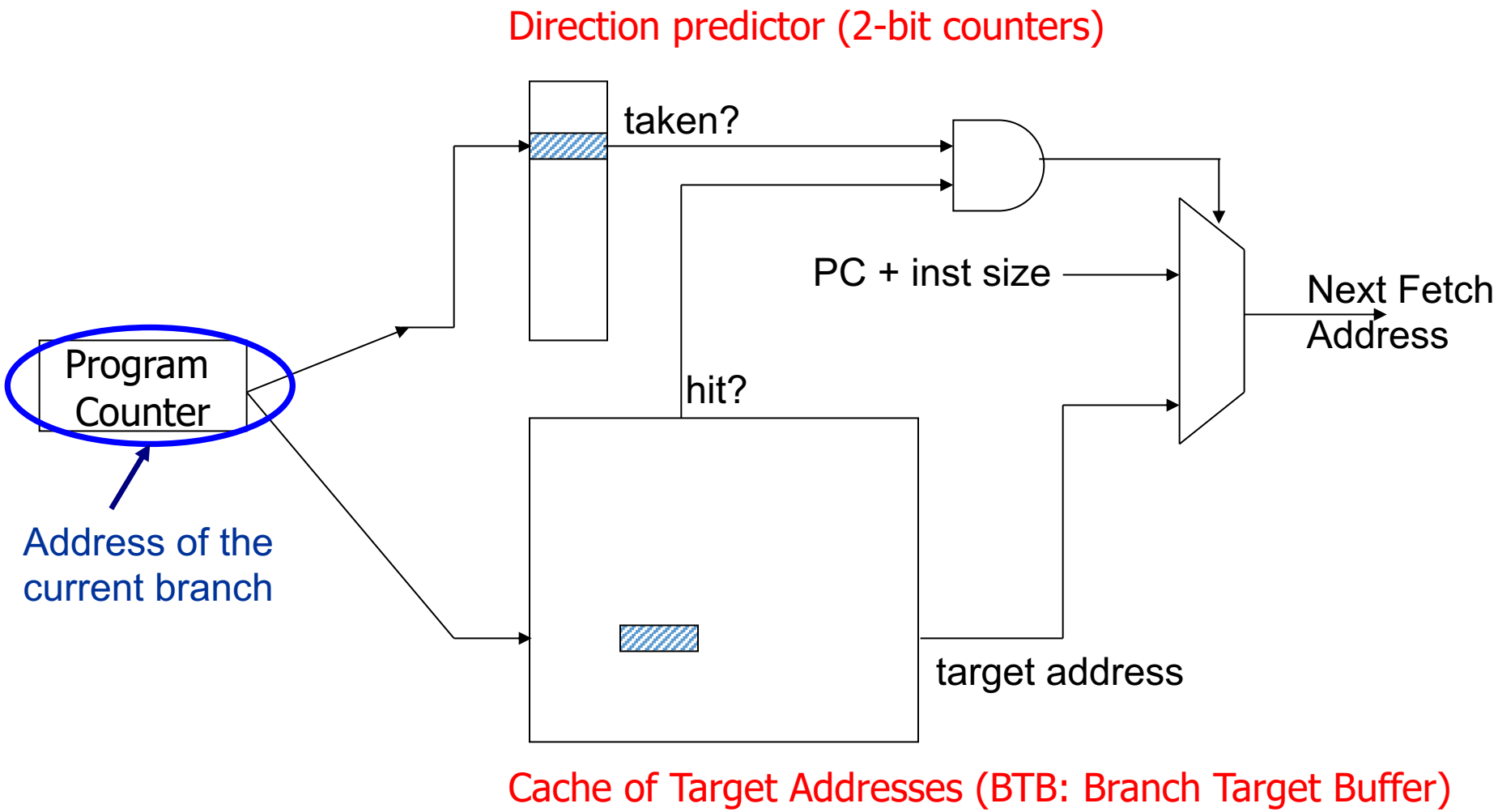
Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early



CPI = [1 + (0.2*0.7) * 1] = 1.14

Fetch Stage with BTB and Direction Prediction



Always taken CPI = [1 + (0.20*0.3) * 2] = 1.12 (70% of branches taken)

More Sophisticated Branch Direction Prediction