Handling Control Dependency -Can we know where to fetch?

Seehwan Yoo

Dept. of Mobile Systems Engineering

Dankook University

DKU DANKOOK UNIVERSITY

Review: data dependency

- comes from register used by instructions in the current pipeline
- WAR, WAW dependency (anti/output dependency) can be resolved by register renaming
  - score board
- Flow dependency (RAW dependency)
  - needs special touch

Review: Resolving flow dependency

- Five ideas
  - software-based interlocking
    - putting nop in software
    - compiler does this
  - hardware-based interlocking (stall)
    - detect and wait
    - pipeline bubbles inserted
  - register forwarding (bypassing)
    - insert additional combinational logic
  - predict the value and verify
    - speculative execution
  - do something else
    - fine-grained multi-threading

Review: Register Forwarding

- dependency check
  - registers in use valid → invalid
  - At decoding stage, we know which registers are accessed
    - if invalid registers are in use, fetch the result value from later pipeline stages
    - if distance is 1, fetch result from
      - EXE stage
    - if distance is 2, fetch result from
      - MEM stage
    - if distance is 3, fetch result from
      - WB stage
      - we can ignore this case by ordering
        - Regfile read (ID stage) after Regfile write (WB stage)

Review: Register Forwarding

- Continuous forwarding
  - what if the case,
    - add r2, r2, r3
    - add r2, r2, r3
    - add r2, r2, r3
- Load-store RAW dependency
  - consider the case
    - lw r2, r1(0)
    - add r3, r2, r4
  - add operation cannot avoid pipeline stall
  - To handle that,
    - MIPS defines delayed load slot

Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction

- 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?

Jump Types

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

Different branch types can be handled differently

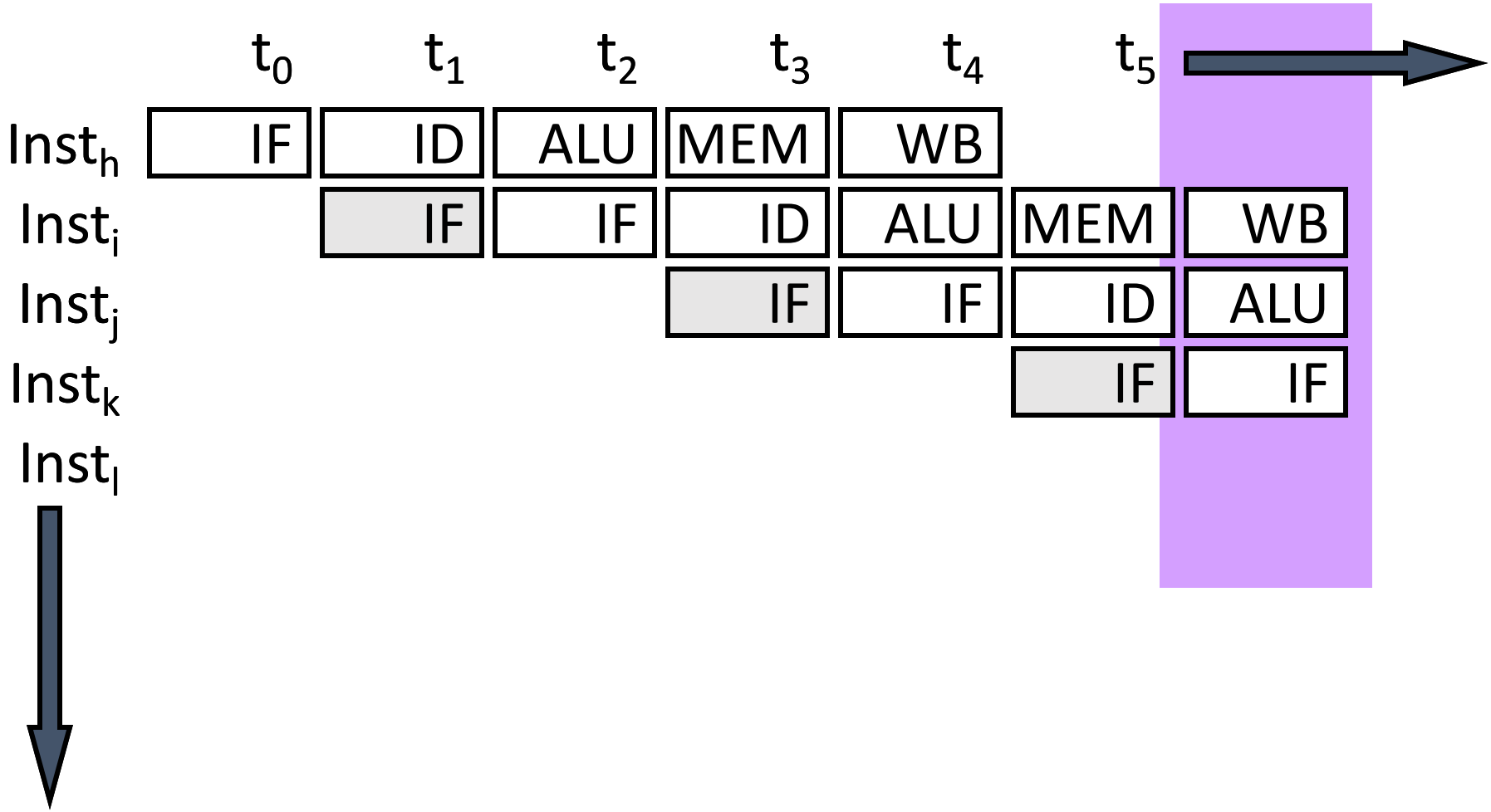
How to Handle Control Dependences

- 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)

Stall Fetch Until Next PC is Available: Good Idea?



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

Doing Better than Stalling Fetch …

- Rather than waiting for true-dependence on PC to resolve, just guess nextPC = PC+4 to keep fetching every cycle
  Is this a good guess?

  What do you lose if you guessed incorrectly?

- ~20% of the instruction mix is control flow
  - ~50 % of "forward" control flow (i.e., if-then-else) is taken
  - ~90% of "backward" control flow (i.e., loop back) is taken

  Overall, typically ~70% taken and ~30% not taken
  [Lee and Smith, 1984]

- Expect "nextPC = PC+4" ~86% of the time, but what about the remainin g 14%?

Guessing NextPC = PC + 4

- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of next fetch address prediction and branch prediction

- How can you make this more effective?

- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed
  - Software: Lay out the control flow graph such that the "likely next instruction" is on the not-taken path of a branch
  - Hardware: ??? (how can you do this in hardware…)

Guessing NextPC = PC + 4

- How else can you make this more effective?

- Idea: Get rid of control flow instructions (or minimize their occurrence)

- How?

  1. Get rid of unnecessary control flow instructions → combine predicates (predicate combining)
  2. Convert control dependences into data dependences → predicated execution

Predicate Combining (*not* Predicated Execution)

- Complex predicates are converted into multiple branches
  - if ((a == b) && (c < d) && (a > 5000)) { … }
    - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction instead of having one branch for each
  - Predicates stored and operated on using condition registers
  - A single branch checks the value of the combined predicate

+ Fewer branches in code → fewer mipredictions/stalls

-- Possibly unnecessary work

-- If the first predicate is false, no need to compute other predicates

- Condition registers exist in IBM RS6000 and the POWER architecture

Predicated Execution

- Idea: Convert control dependence to data dependence

- Suppose we had a Conditional Move instruction…
  - CMOV condition, R1 ← R2
  - R1 = (condition == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)

- Code example with branches vs. CMOVs

  if (a == 5) {b = 4;} else {b = 3;}

  CMPEQ condition, a, 5;
  CMOV condition, b ← 4;
  CMOV !condition, b ← 3;

Predicated Execution

- Eliminates branches → enables straight line code (i.e., larger basic blocks in code)

- Advantages
  - Always-not-taken prediction works better (no branches)
  - Compiler has more freedom to optimize code (no branches)
    - control flow does not hinder inst. reordering optimizations
    - code optimizations hindered only by data dependencies

- Disadvantages
  - Useless work: some instructions fetched/executed but discarded (especially bad for easy-to-predict branches)
  - Requires additional ISA support

- Can we eliminate all branches this way?

Predicated Execution

- We will get back to this…

- Some readings (optional):
  - Allen et al., "Conversion of control dependence to data dependence," POPL 1983.
  - Kim et al., "Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution," MICRO 2005.

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 (I)

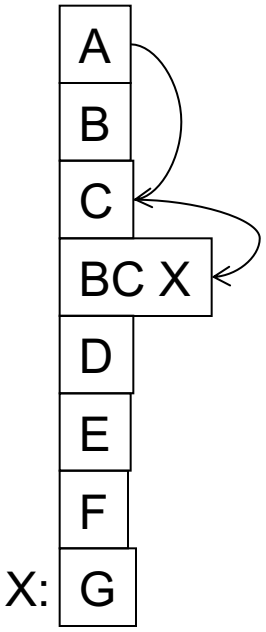
- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles
- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are always executed regardless of branch direction.

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

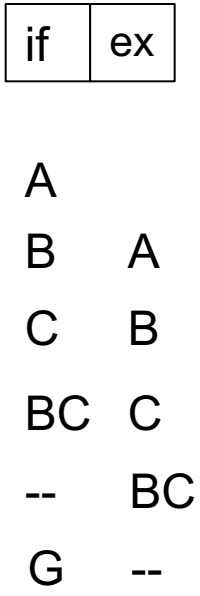
- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot

Delayed Branching (II)

Normal code:

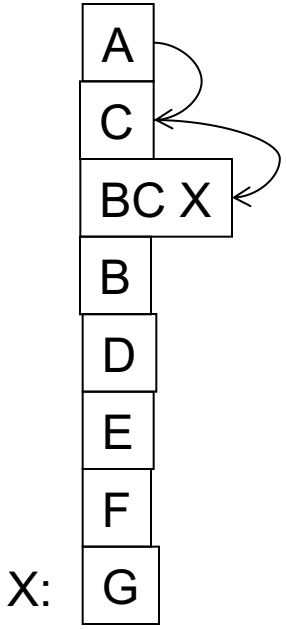


Timeline:

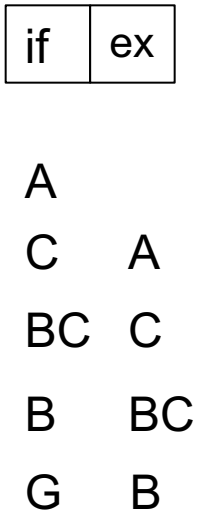


6 cycles

Delayed branch code:



Timeline:

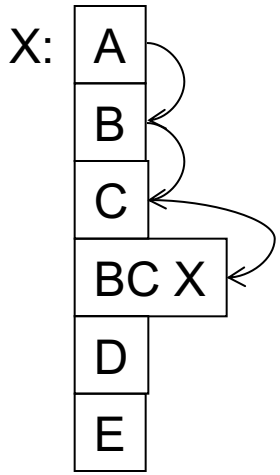


5 cycles

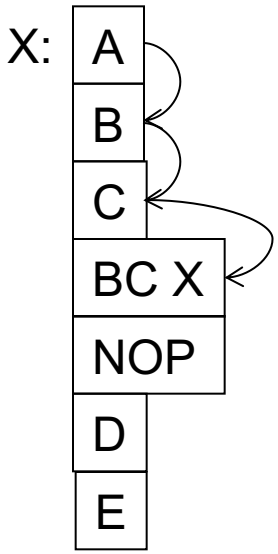
Fancy Delayed Branching (III)

- Delayed branch with squashing
  - In SPARC
  - If the branch falls through (not taken), the delay slot instruction is not executed
  - Why could this help?

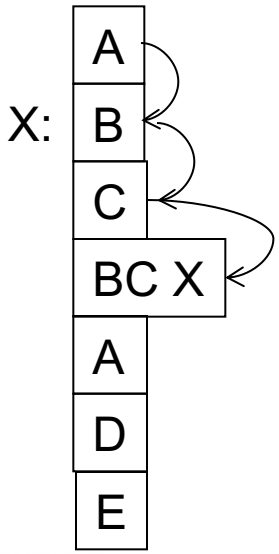
Normal code:



Delayed branch code:



Delayed branch w/ squashing:



Delayed Branching (IV)

- Advantages:

  + Keeps the pipeline full with useful instructions in a simple way assuming

    1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves

    2. All delay slots can be filled with useful instructions

- Disadvantages:

  -- Not easy to fill the delay slots (even with a 2-stage pipeline)

    1. Number of delay slots increases with pipeline depth, superscalar execution width

    2. Number of delay slots should be variable with variable latency operations. Why?

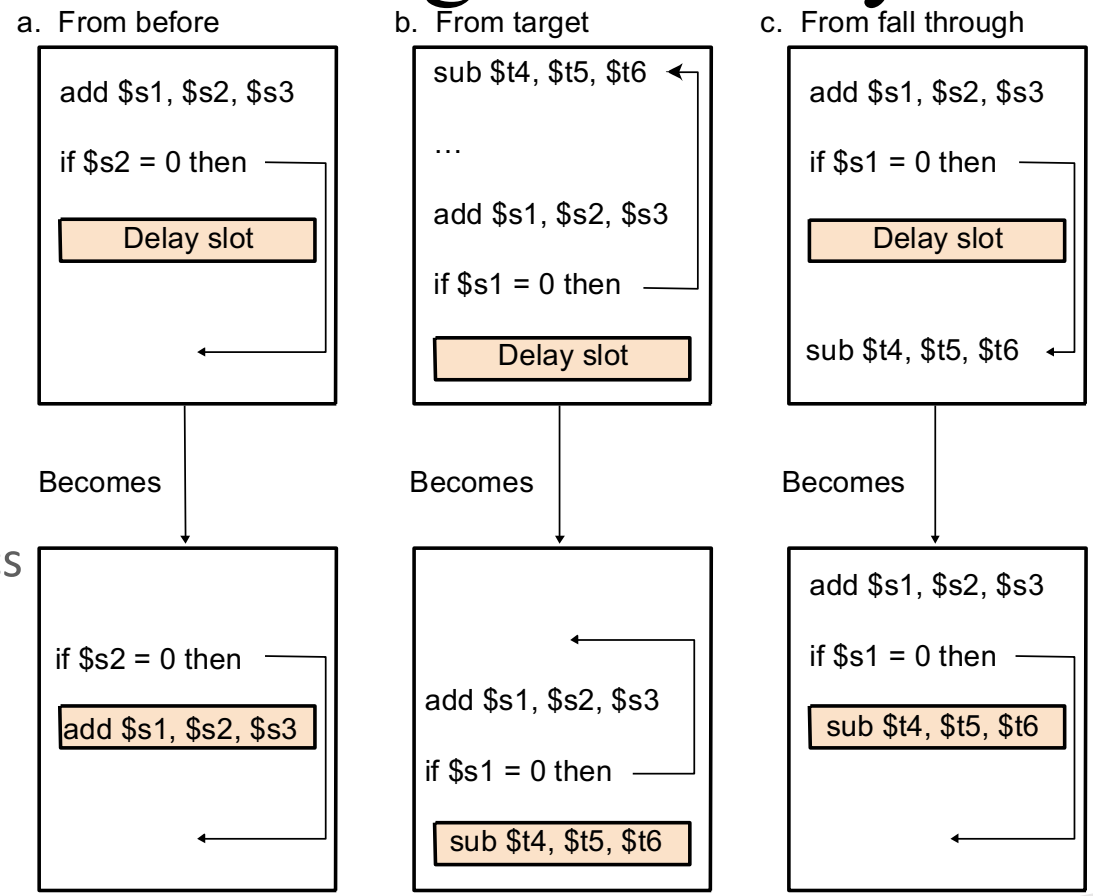
  -- Ties ISA semantics to hardware implementation

    -- SPARC, MIPS, HP-PA: 1 delay slot

    -- What if pipeline implementation changes with the next design?

An Aside: Filling the Delay Slot

reordering data independent (RAW, WAW, WAR) instructions does not change program semantics



within same basic block

a new instruction added to not-taken path??

a new instruction added to taken??

Safe?

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)

Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts. Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, there is no need to fetch another instruction from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads' instructions

+ No logic needed for handling control and data dependences within a thread

-- Single thread performance suffers

-- Extra logic for keeping thread contexts

-- Does not overlap latency if not enough threads to cover the whole pipeline

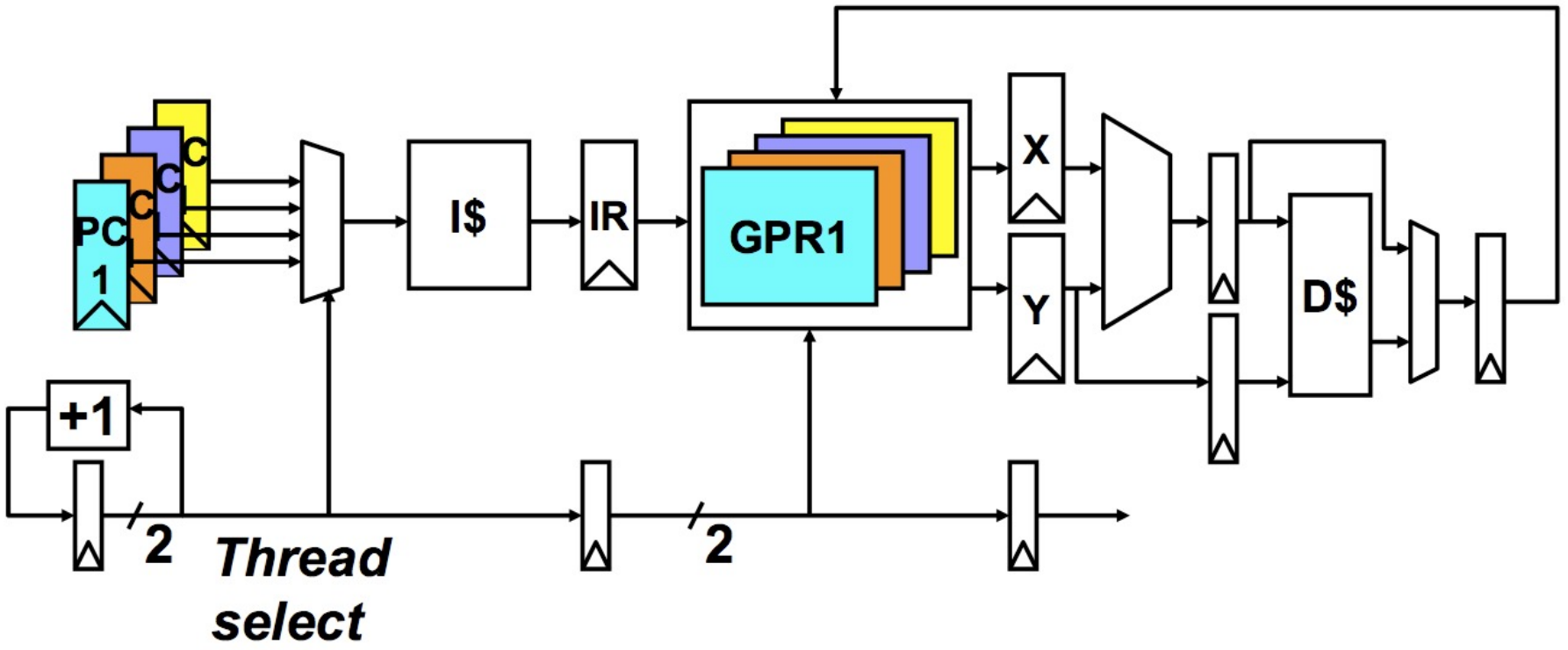
Fine-grained Multithreading

- Idea: Switch to another thread every cycle such that no two instructions from the thread are in the pipeline concurrently

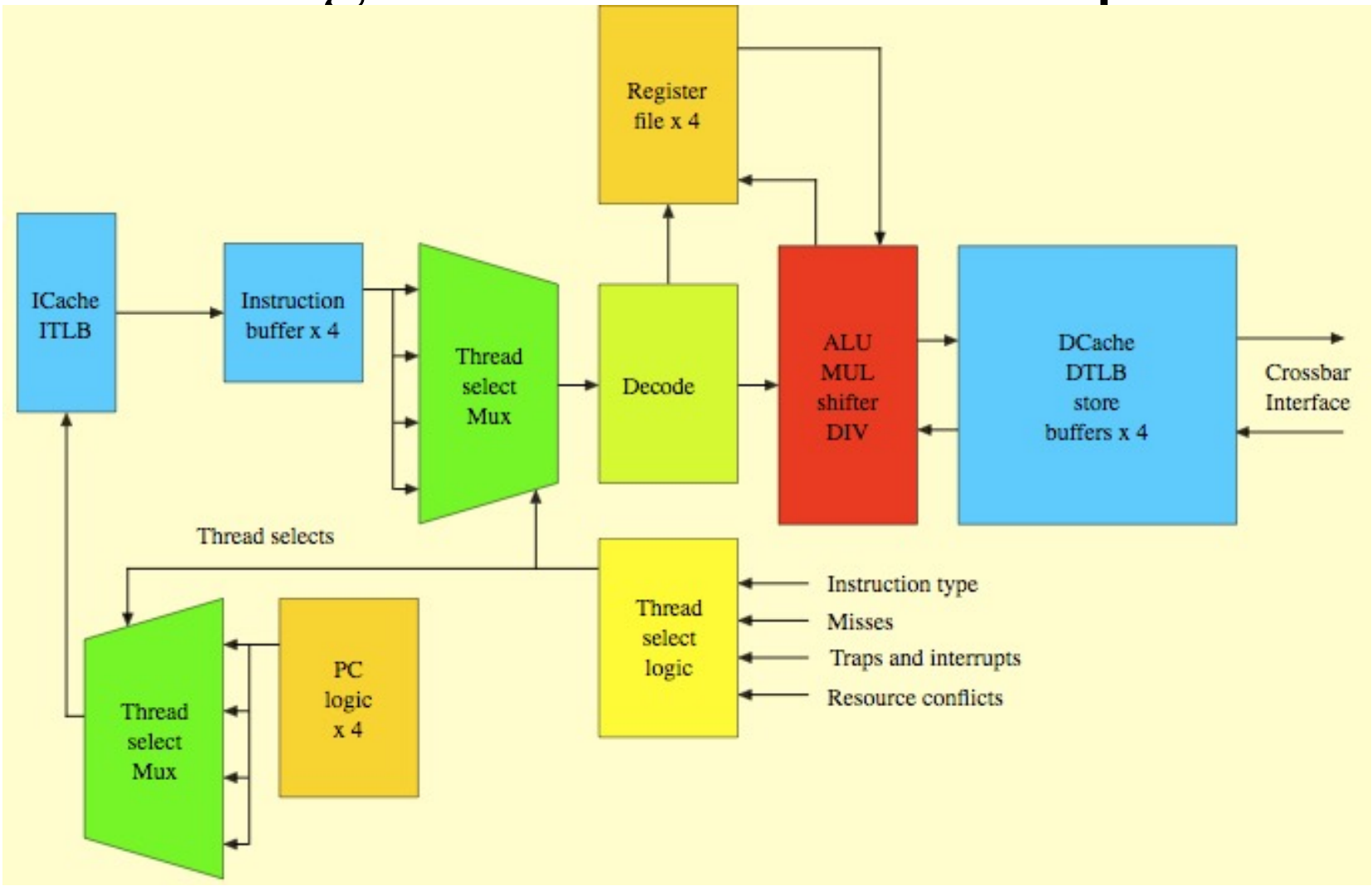
- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads

- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

Multithreaded Pipeline Example



Sun Niagara Multithreaded Pipeline



Fine-grained Multithreading

- Advantages
  - + No need for dependency checking between instructions
    (only one instruction in pipeline from a single thread)
  - + No need for branch prediction logic
  - + Otherwise-bubble cycles used for executing useful instructions from different threads
  - + Improved system throughput, latency tolerance, utilization

- Disadvantages
  - - Extra hardware complexity: multiple hardware contexts, thread selection logic
  - - Reduced single thread performance (one instruction fetched every N cycles)
  - - Resource contention between threads in caches and memory
  - - Some dependency checking logic between threads remains (load/store)

Brach prediction
A sophisticated instruction fetch – in modern cpus

Seehwan Yoo

Spring 2015

DKU.CIS.MSE

DANKOOK UNIVERSITY

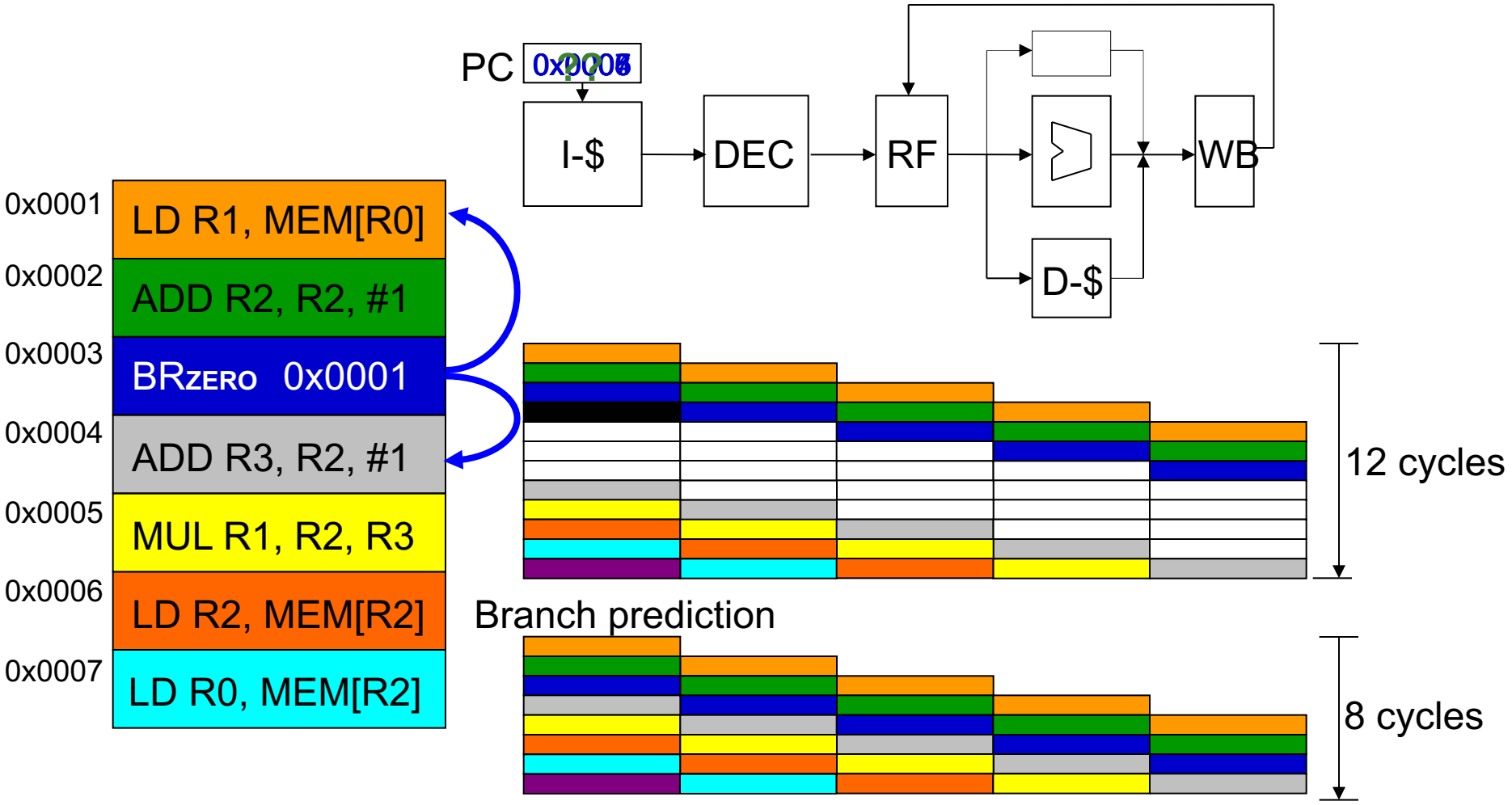
How to Handle Control Dependences

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

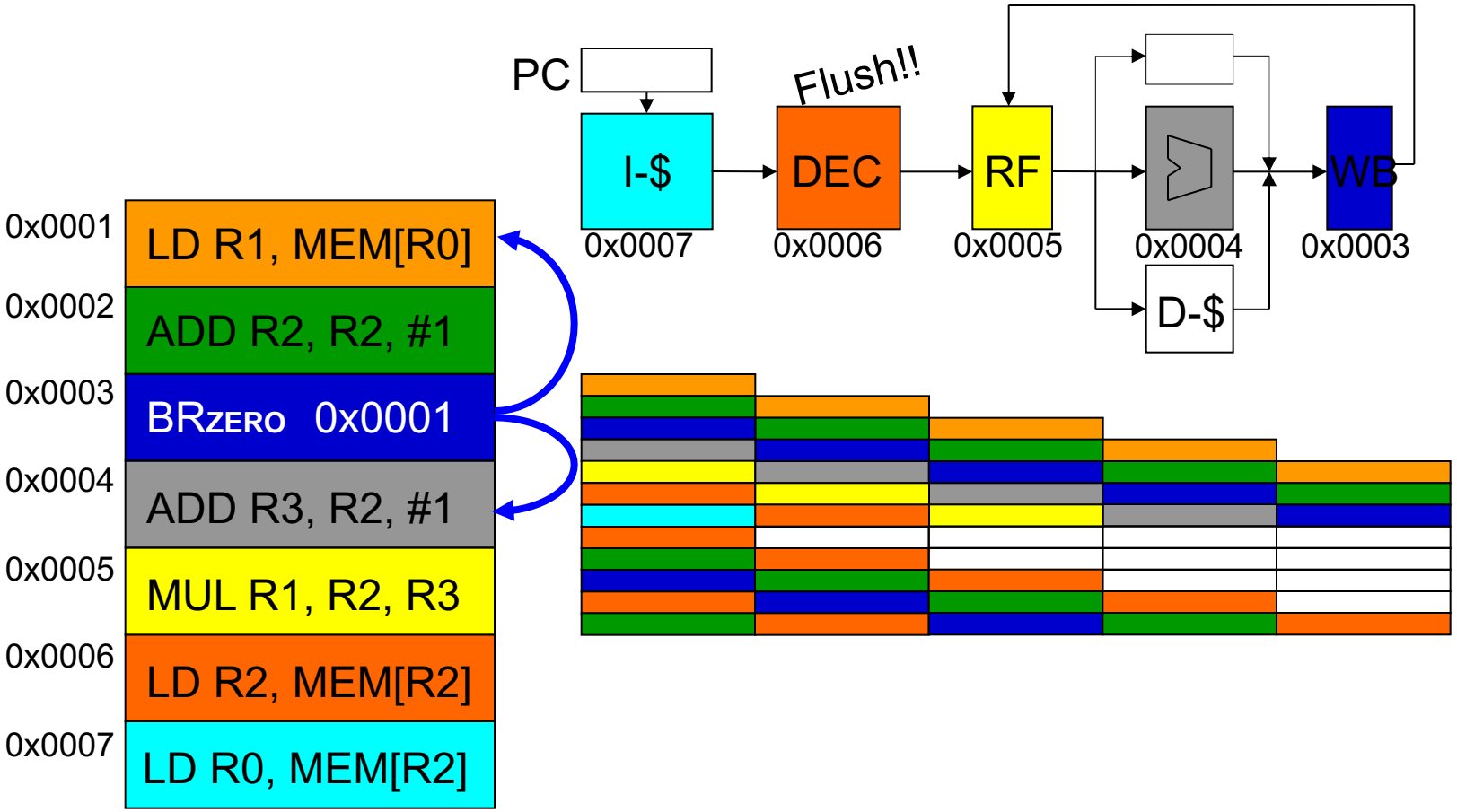
- Potential solutions if the instruction is a control-flow instruction:

- 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: Guess the Next Instruction to Fetch

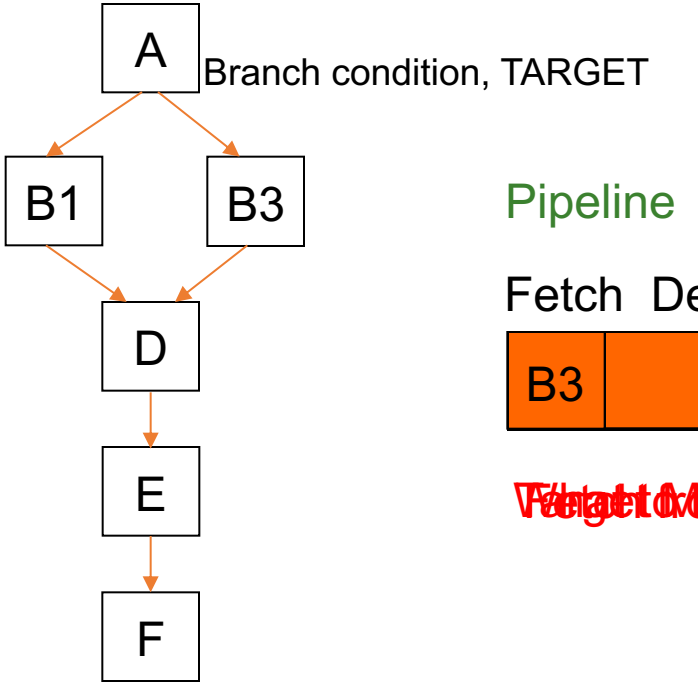


Misprediction Penalty

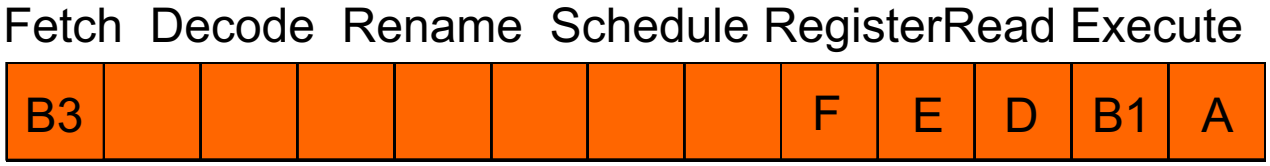


Branch Prediction

- 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

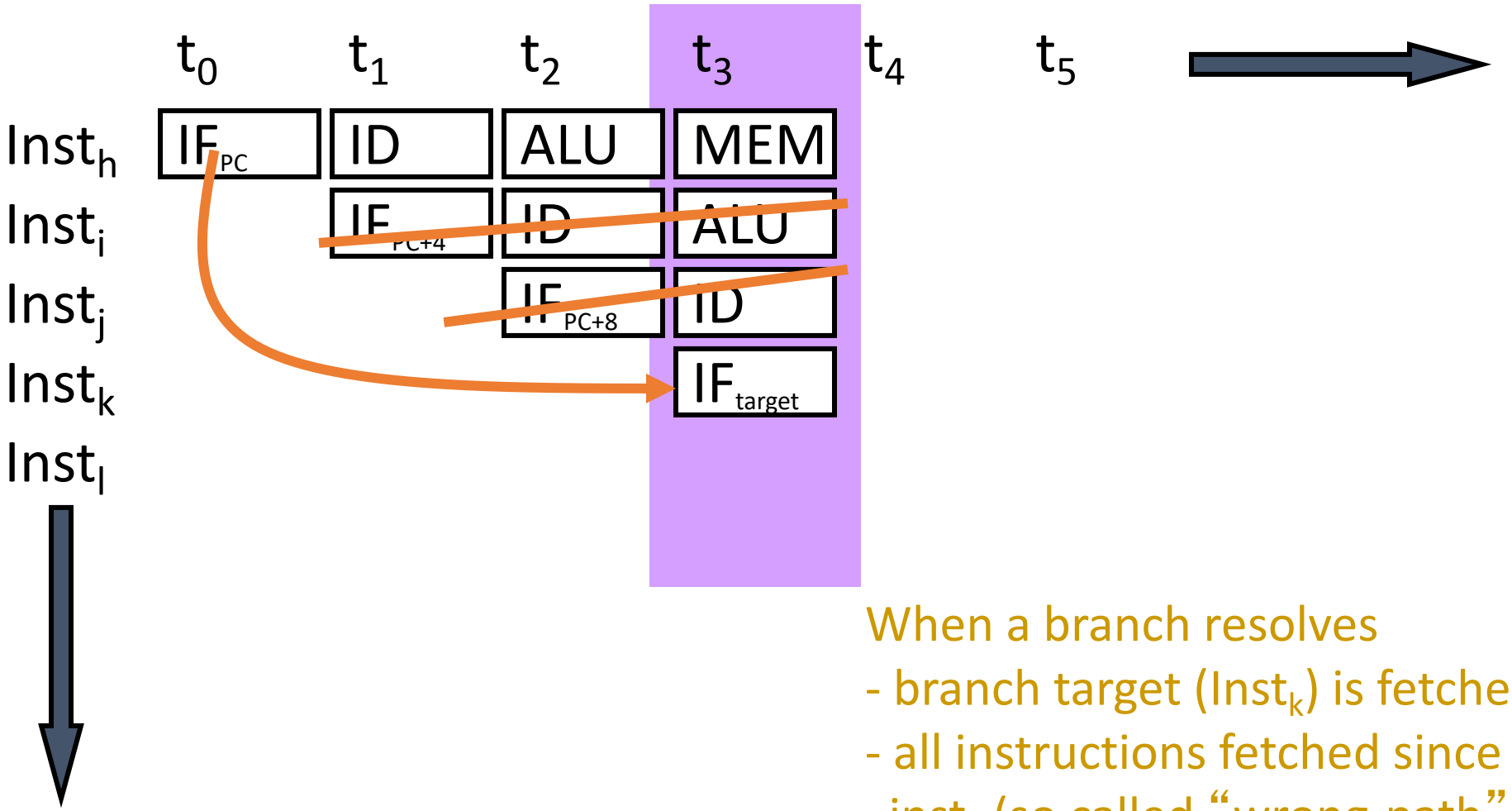


Pipeline



Target Misprediction Detected! Flush the Pipeline

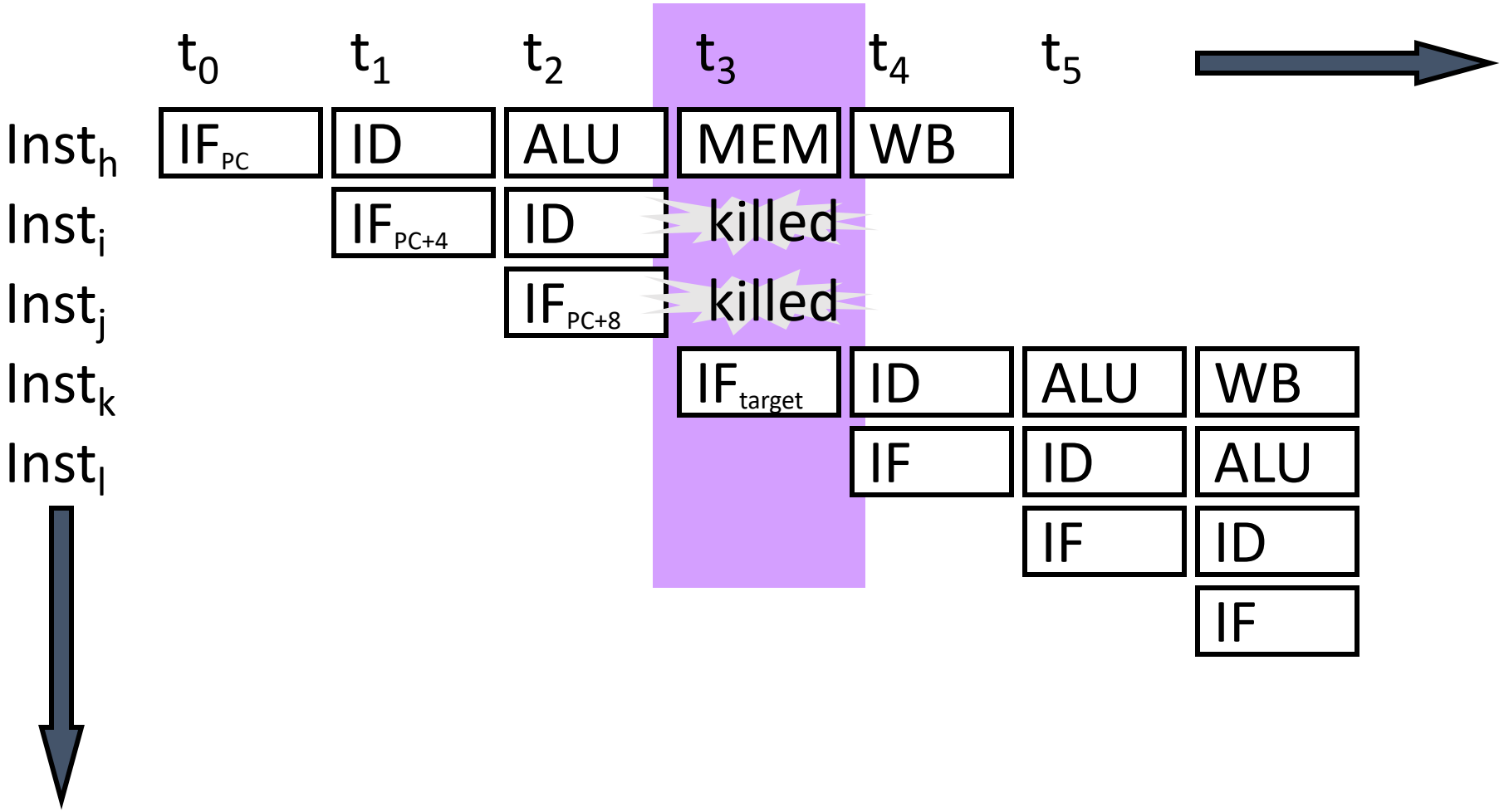
Branch Prediction: Always PC+4



Insth is a branch

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

Pipeline Flush on a Misprediction



$Inst_h$ 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

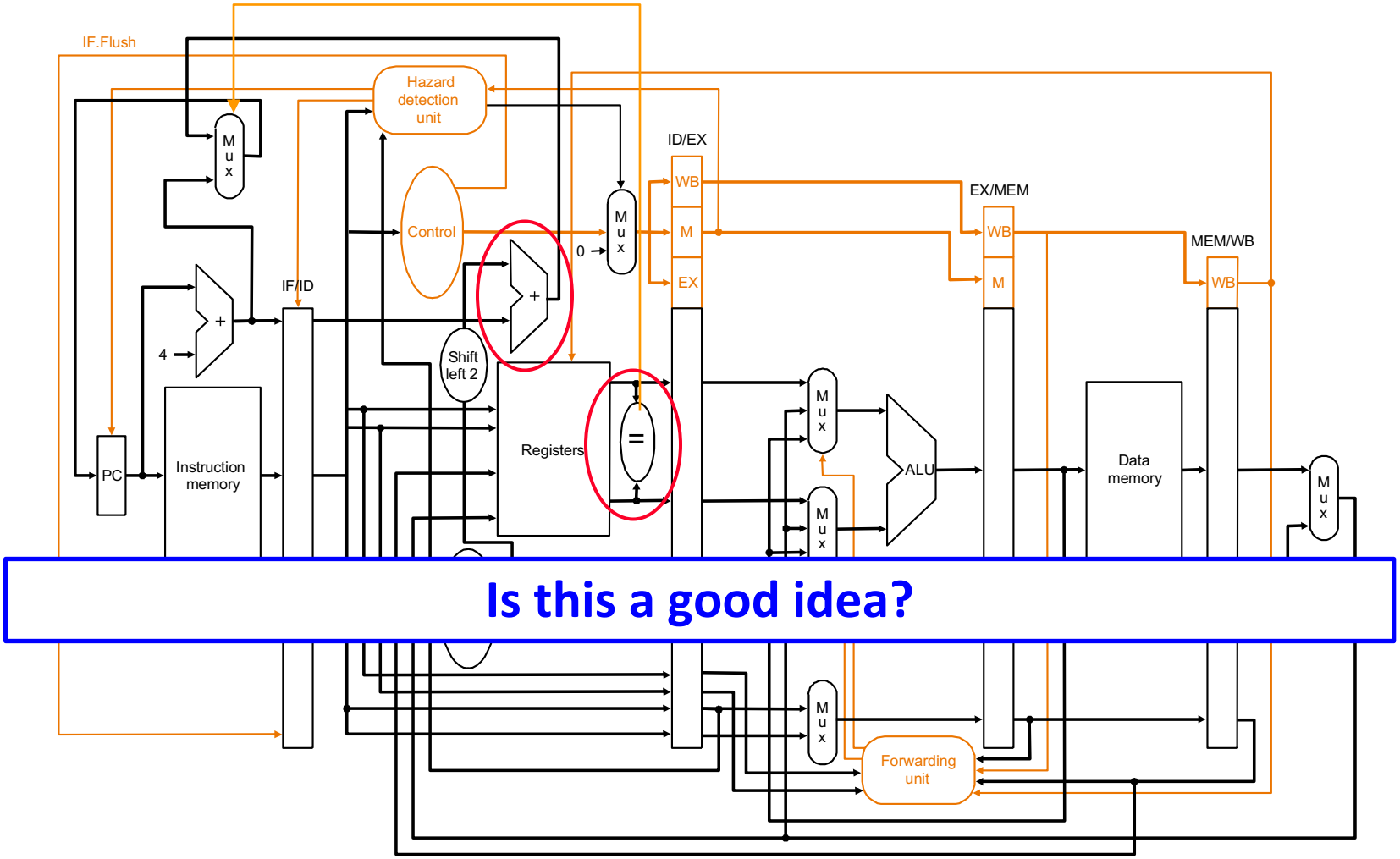
probability of a wrong guess

penalty for a wrong guess

Can we reduce either of the two penalty terms?

Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early



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

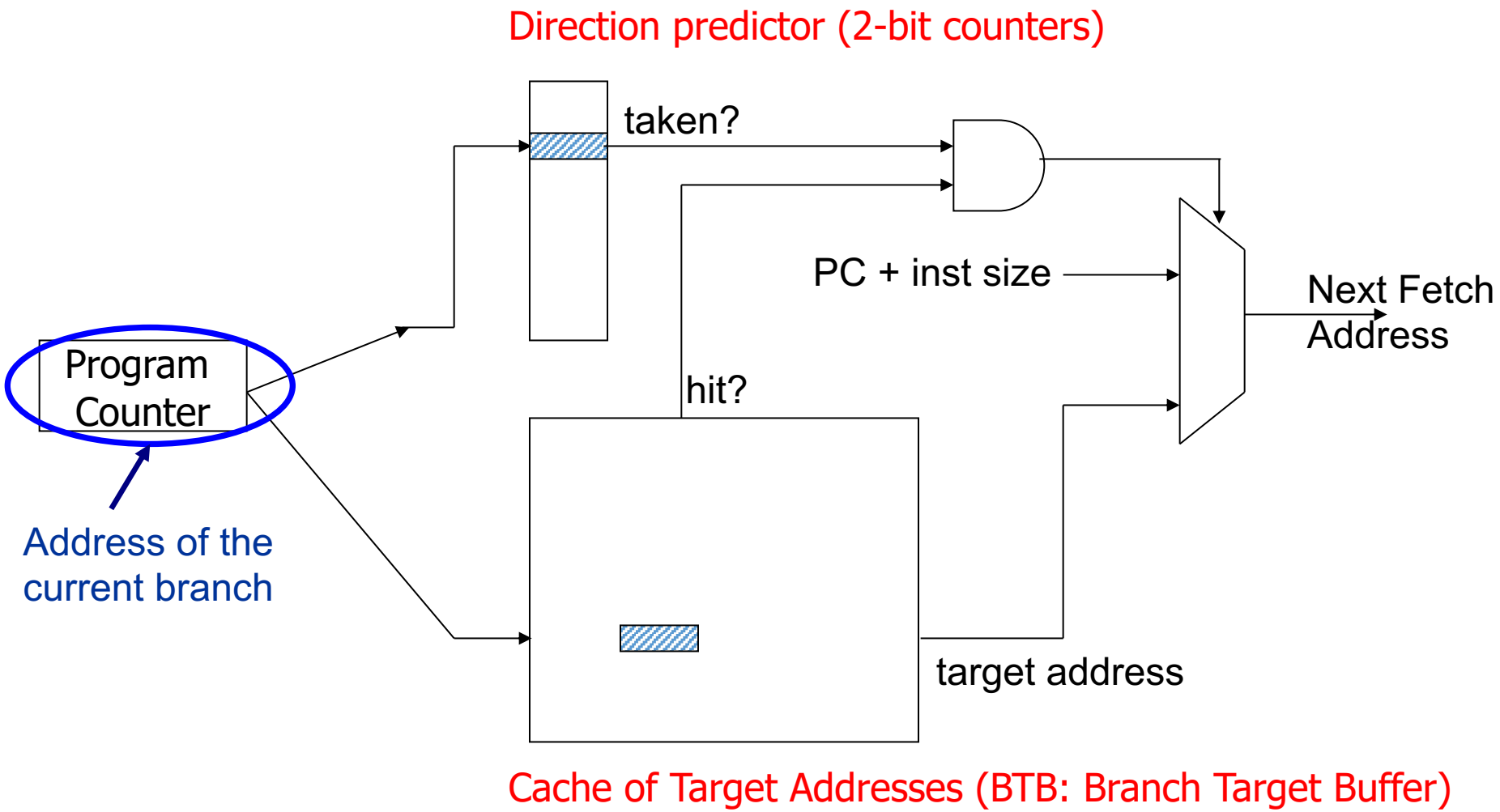
Branch Prediction (Enhanced)

- 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)

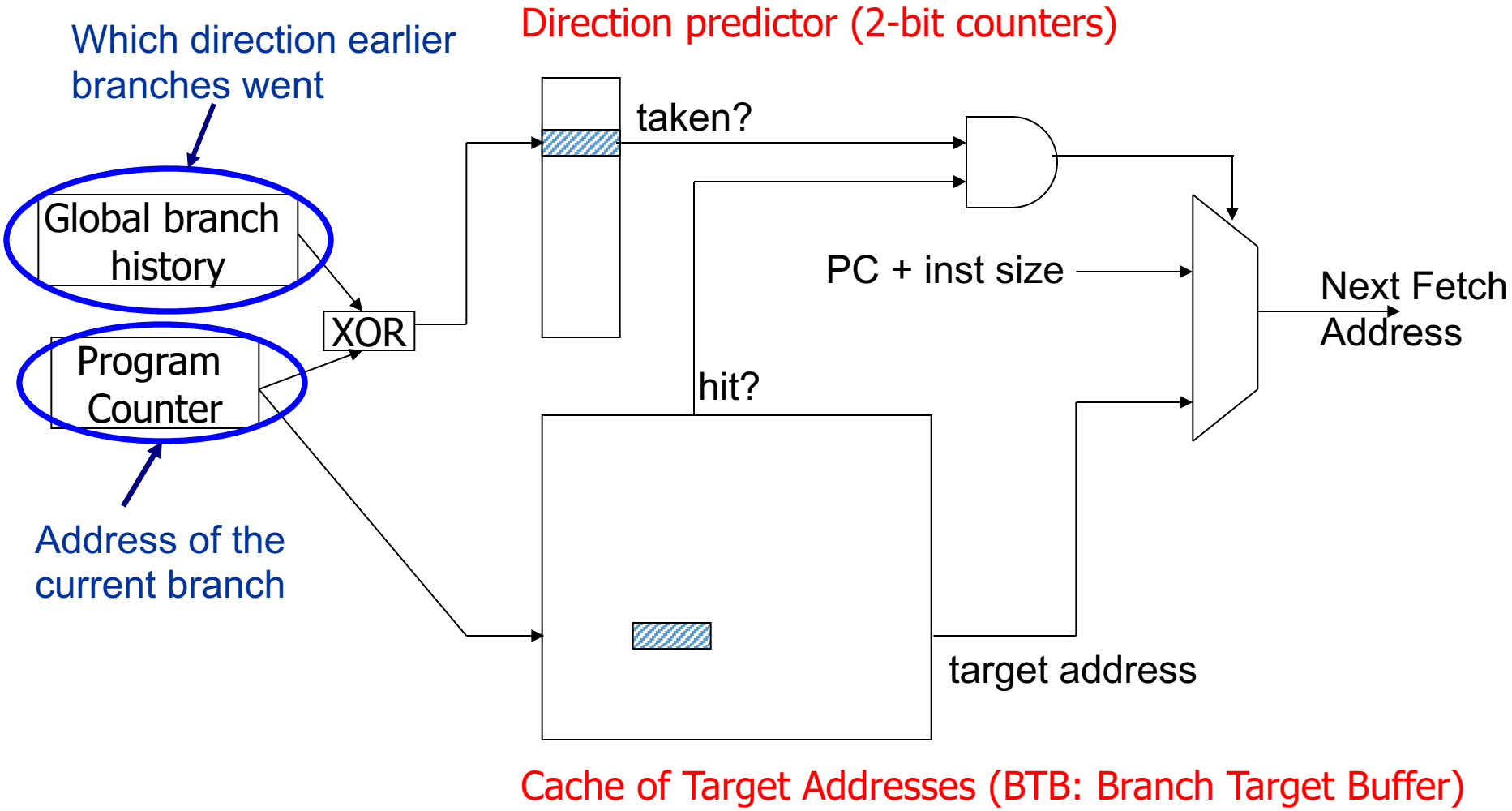
- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - Idea: Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

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



Simple Branch Direction Prediction Schemes

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

- Run time (dynamic)
  - Last time prediction (single-bit)

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