Pipelining

Prof Rajeev Barua

E-A 001 -- Slide set 4

What is pipelining?

Pipelining is a way to start the next instruction before the current instruction is finished.

Analogy of car assembly line:

Henry Ford with Model T in the 1910's

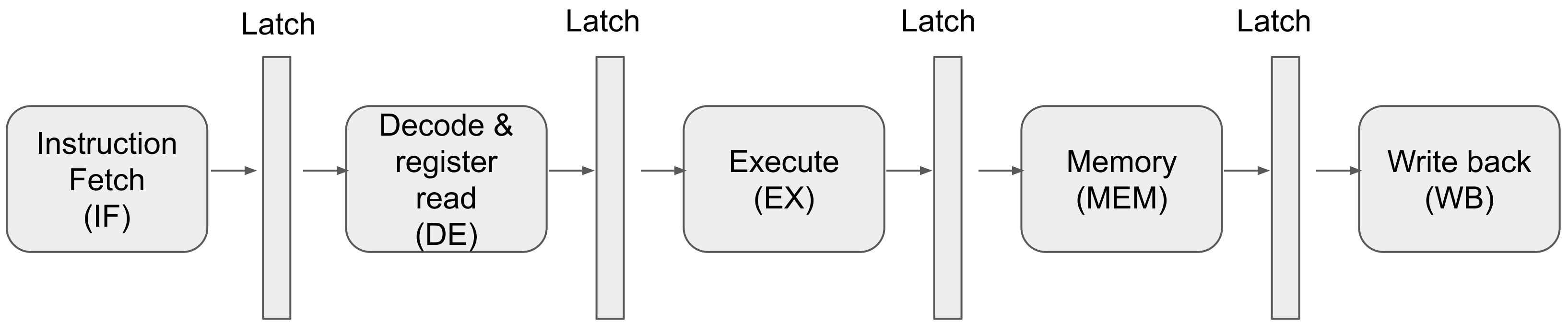
- Building a car is broken up into stations, each accomplishing one task.
- Dedicated crew at each station.
- Car moves to next station when done.

Advantages vs serial assembly:

- Crew gets specialized to one task ⇒ they become faster.
    - Since the tasks are now simple, they could be automated in some cases.
- A new car in time for one step, instead times for all steps combined.

Pipelining in a modern computer

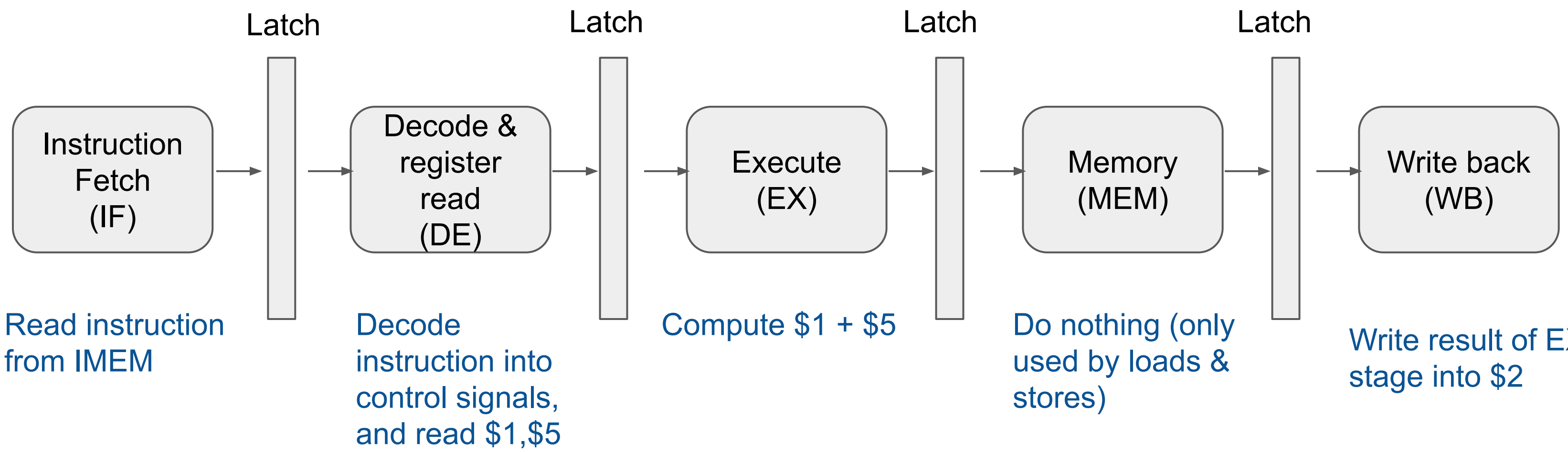
MIPS pipeline:



- Above is the classic 5-stage pipeline. Used in MIPS.
- Each stage does one task for running an instruction
- Each stage takes 1 clock cycle to run
- Latches store and hold the result of the previous stage for 1 cycle, so next stage has stable inputs to compute on.

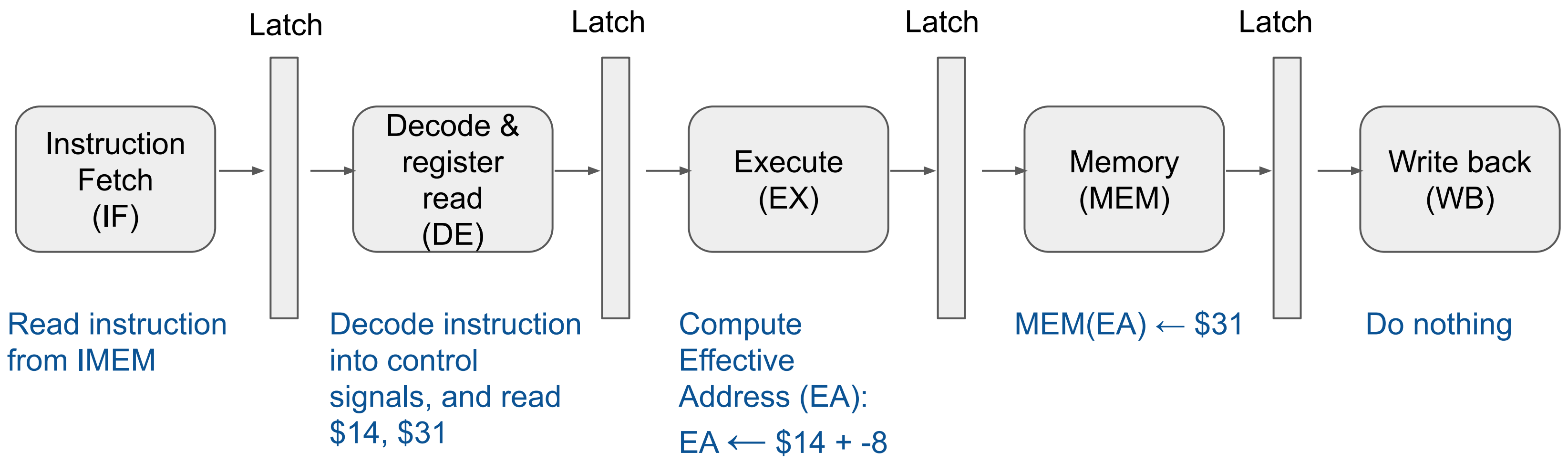
Example tasks in an add instruction

Example: add $2, $1, $5


Latch
Latch
Latch
Latch
Instruction Fetch (IF)
Decode & register read (DE)
Execute (EX)
Memory (MEM)
Write back (WB)
Read instruction from IMEM
Decode instruction into control signals, and read $1,$5
Compute $1 + $5
Do nothing (only used by loads & stores)
Write result of EX stage into $2

Example tasks in a Store instruction

Example: sw $31, -8($14)
Semantics: MEM($14 + -8) ⟵ $31



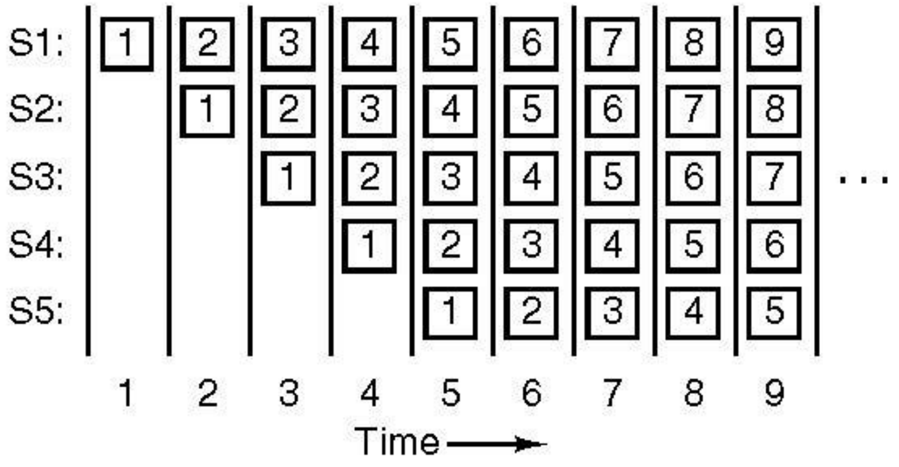
Latch
Latch
Latch
Latch
Instruction Fetch (IF)
Decode & register read (DE)
Execute (EX)
Memory (MEM)
Write back (WB)
Read instruction from IMEM
Decode instruction into control signals, and read $14, $31
Compute Effective Address (EA): EA ⟵ $14 + -8
MEM(EA) ⟵ $31
Do nothing

Timing of pipeline tasks

Instructions in a pipeline start before the previous instruction finishes.

Whole idea is to complete one instruction in 1 cycle, instead of in 5 cycles.

- The key to doing this is to overlap work across stages for different instructions



S1, … S5 are the five stages in the MIPS pipeline

Pipelining is ubiquitious today

All modern processors are pipelined.

Modern processors have very deep pipelines.

- Advantage:
  - shorter time per stage ⇒ greater initiation rate
- Disadvantage:
  - dependences stalls become more expensive. (discussed next)

Complications in pipelining

Complications: Unlike in Assembly line.

There are dependencies between instructions: (Very complicated topic!!!)

- Data dependencies.
- Control dependencies

Data dependencies

```
add    $3, $2, $7
sub    $5, $3, $9
```

Problem is:

- The result of add (**$**3) is used by the sub instruction.
- But when sub reads $3 in DE stage, add is in EX add has not yet written result to $3!

A simple solution:

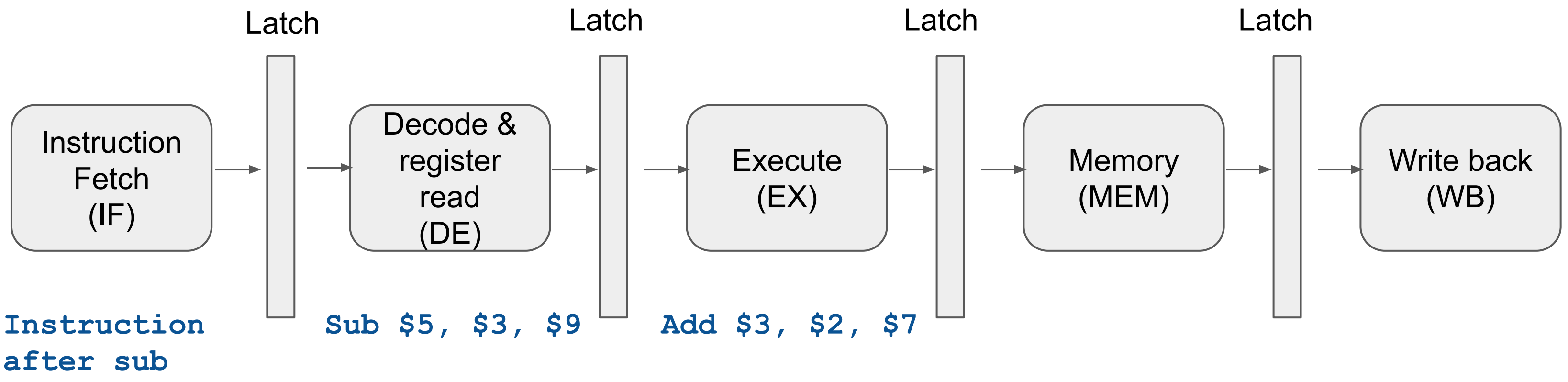
- Stall the sub instruction in the DE stage for 2 cycles.

Data dependence: why there is a problem

```
add     $3, $2, $7
sub     $5, $3, $9
```

The problem scenario:

- When the sub instruction reads $3 in DE, the add instruction has not written $3 yet.



Reads $3 in second half of cycle

Has not written $3 yet (will write later in WB stage)!

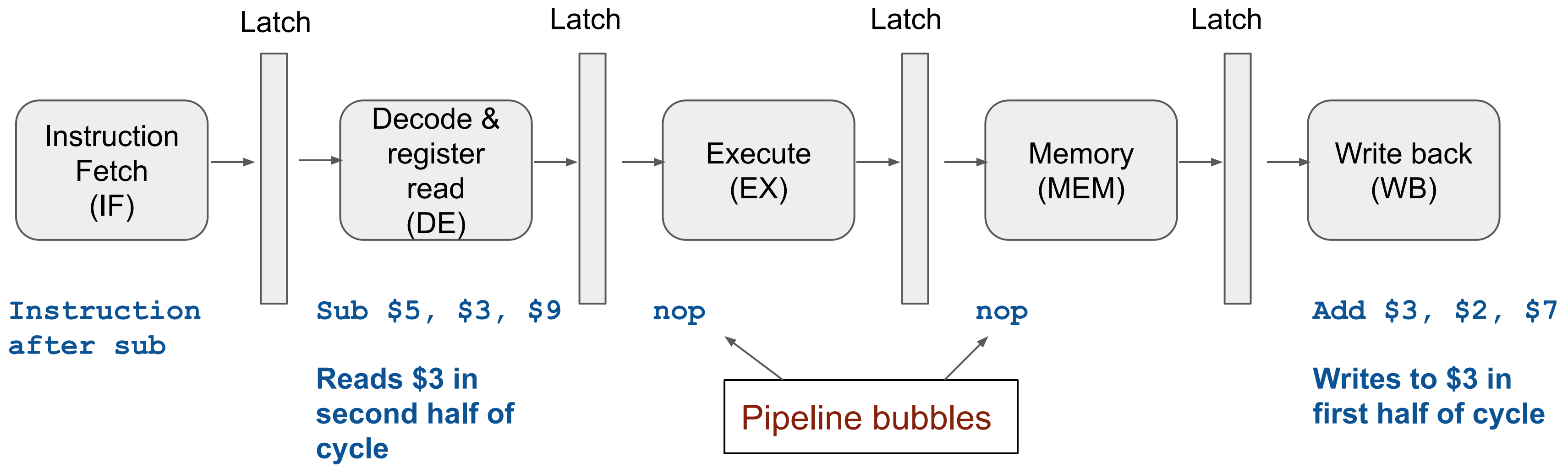
If we do nothing, the sub instruction will read the old value of $3 before the add instruction, leading to incorrect execution.

A simple solution: pipeline stall

```
add    $3, $2, $7
sub    $5, $3, $9
```

A simple solution:

- Stall the sub instruction in the DE stage for 2 cycles:

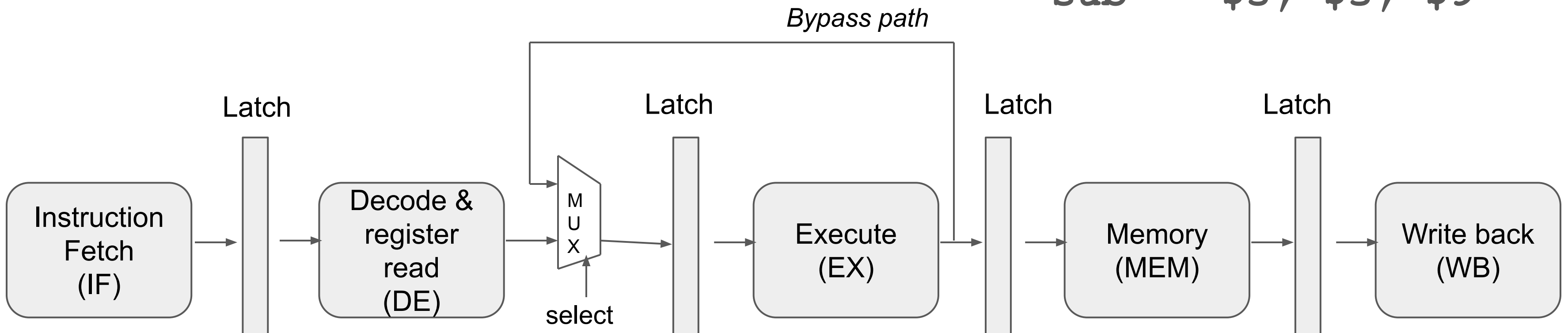


Downside of the above: lose 2 cycles in execution.

A better solution: bypass paths

Feed the result of the EX stage to output of DE stage:

```
add    $3, $2, $7
sub    $5, $3, $9
```



`Instruction after sub`

`Sub $5, $3, $9`

`Add $3, $2, $7`

MUX select input chooses bypass path if data dependence is present

Result $2 + $7 is forwarded to DE stage

No stall cycles ⇒ run at full speed!

Other bypass paths:

- From MEM to DE to the same MUX (when unrelated instruction between add and sub above)
- From MEM to EX (when lw then sw)

Control dependencies

```
          beqz   $3, target
          xor    ...
          ...
target:   and    ...
```

Problem is:

- If we change nothing, then **`beqz`** is followed by **`xor`** in 5 stage pipeline, always.
- That is not correct when branch condition **`($3 == 0)`** is true!

A simple solution:

- Kill the **`xor`** later, once the branch is "resolved", if is found to be taken. Then fetch the **`and`** instruction.
  - This killing of an instruction is called a "pipeline squash".

When is a branch "resolved"?

A branch is said to be resolved when both its target and condition (if any) are computed for this run.

```
          beqz   $3, target
          xor    …
          …
target:   and    …
```

When is a branch resolved in the classic 5-stage MIPS pipeline?

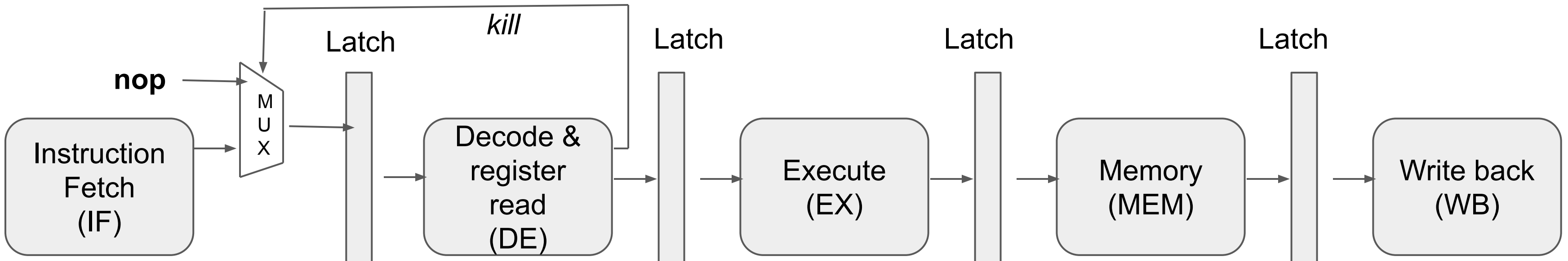
- Without extra hardware, the branch is resolved in the EX stage, when the PC + target calculation is done with the EX stage's ALU. In this case, the condition comparison (if any) can also be done in the EX stage. *(result is 2 cycle stall)*
- With extra hardware, the branch resolution can be done earlier in the DE stage. In this case, an extra adder is needed in the DE stage to perform PC + target, and the condition comparison (if any) can also be done in the DE stage. *(result is 1 cycle stall)*
  - **Caution:** if adding the condition comparison at the end of the DE stage increases the cycle time, then it is better not to, and instead resolve the branch in EX instead.

A simple solution: pipeline squash

```
        beqz $3, target
        xor      …
        …
target: and      …
```

A simple solution:

- Kill the **`xor`** if branch condition is true. Picture below assumes branch resolution in DE.



Cycle 1: ~~`xor`~~ `beqz $3,_target`

Cycle 2: `and` `nop` `beqz $3,_target`

Pipeline bubble

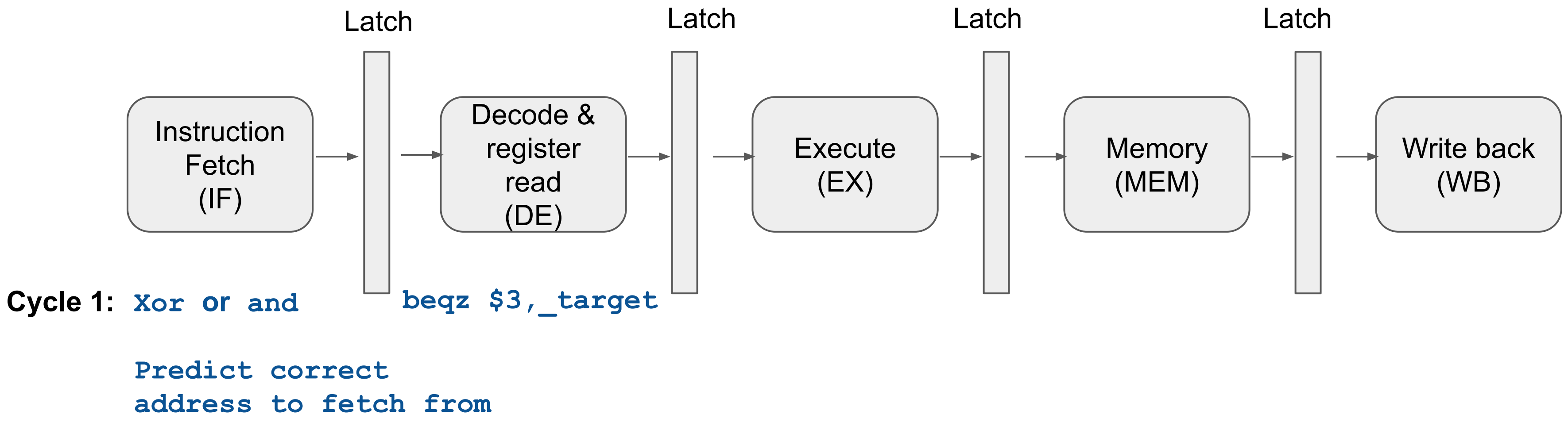
Downside of the above: lose 1 cycles in execution. Called "branch penalty". Will be 2 cycles pipeline bubble if branch resolved in EX instead.

A better solution: branch prediction

```
        beqz    $3, target
        xor     …
        …
target: and     …
```

A better solution:

- Predict the next instruction after branch:



The branch predictor hardware takes PC of instruction in IF stage. Outputs the predicted next address to fetch from in the next cycle. If correct, no stall. If wrong, then squash.

Branch prediction intuition

```
          beqz   $3, target
          xor    …
          …
target:   and    …
```

Intuition:

- Use a history-based predictor called a Branch Target Buffer (BTB).
    - Observe and store what the next address was the last time this branch was executed. Only store all CTIs in BTB. Include unconditional branches as they also need to be predicted, since we don’t know their target without it.
    - Don’t store any prediction for other instructions (same as predict fall through).
- How to implement this is out of syllabus.
- Branch prediction is essential in modern deep pipelines
    - Because branch penalties can be several cycles long,
    - A great deal of effort has gone into making this better!