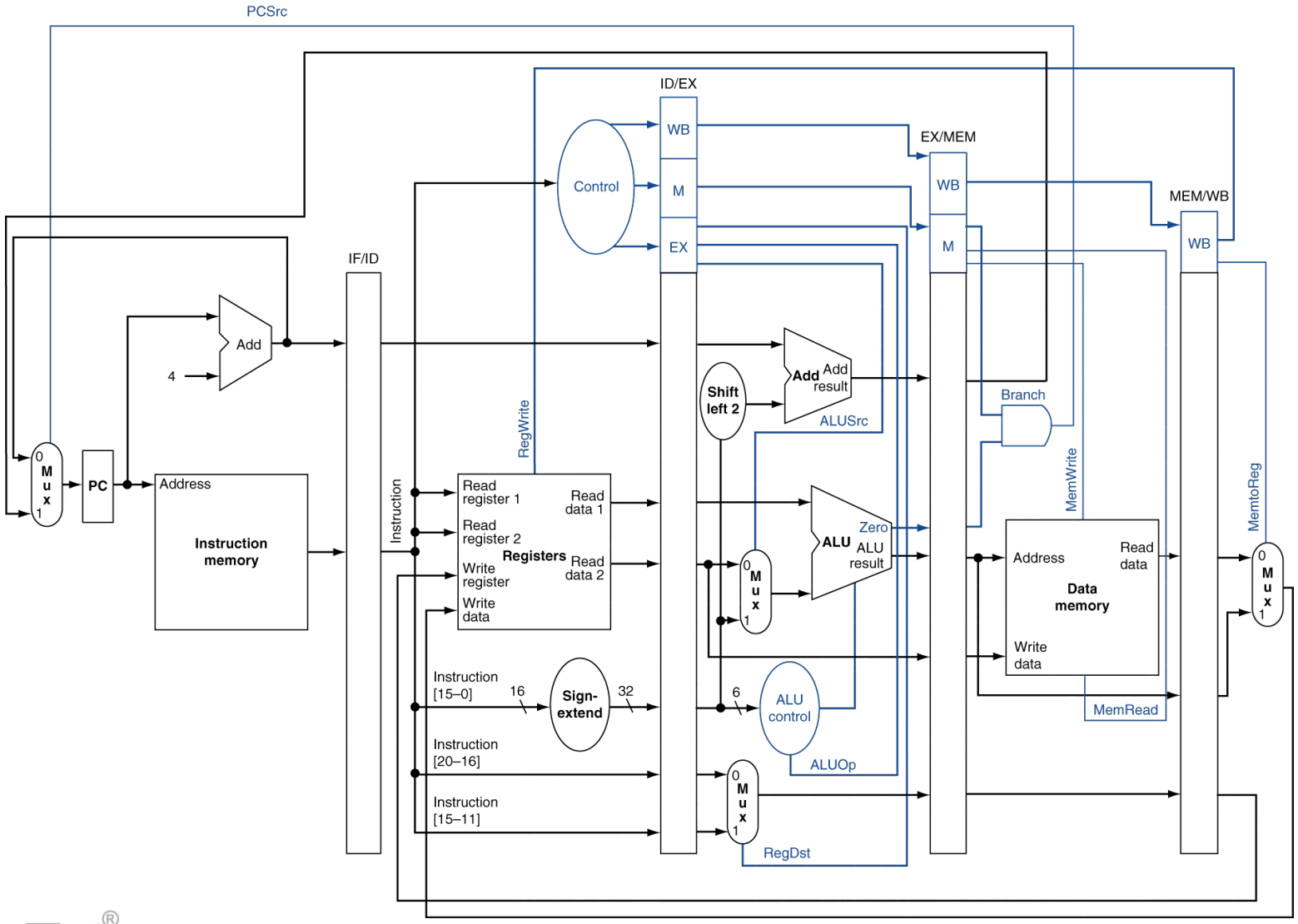
Chapter 4

Brach Hazard: Detailed Analysis

Branch Hazards

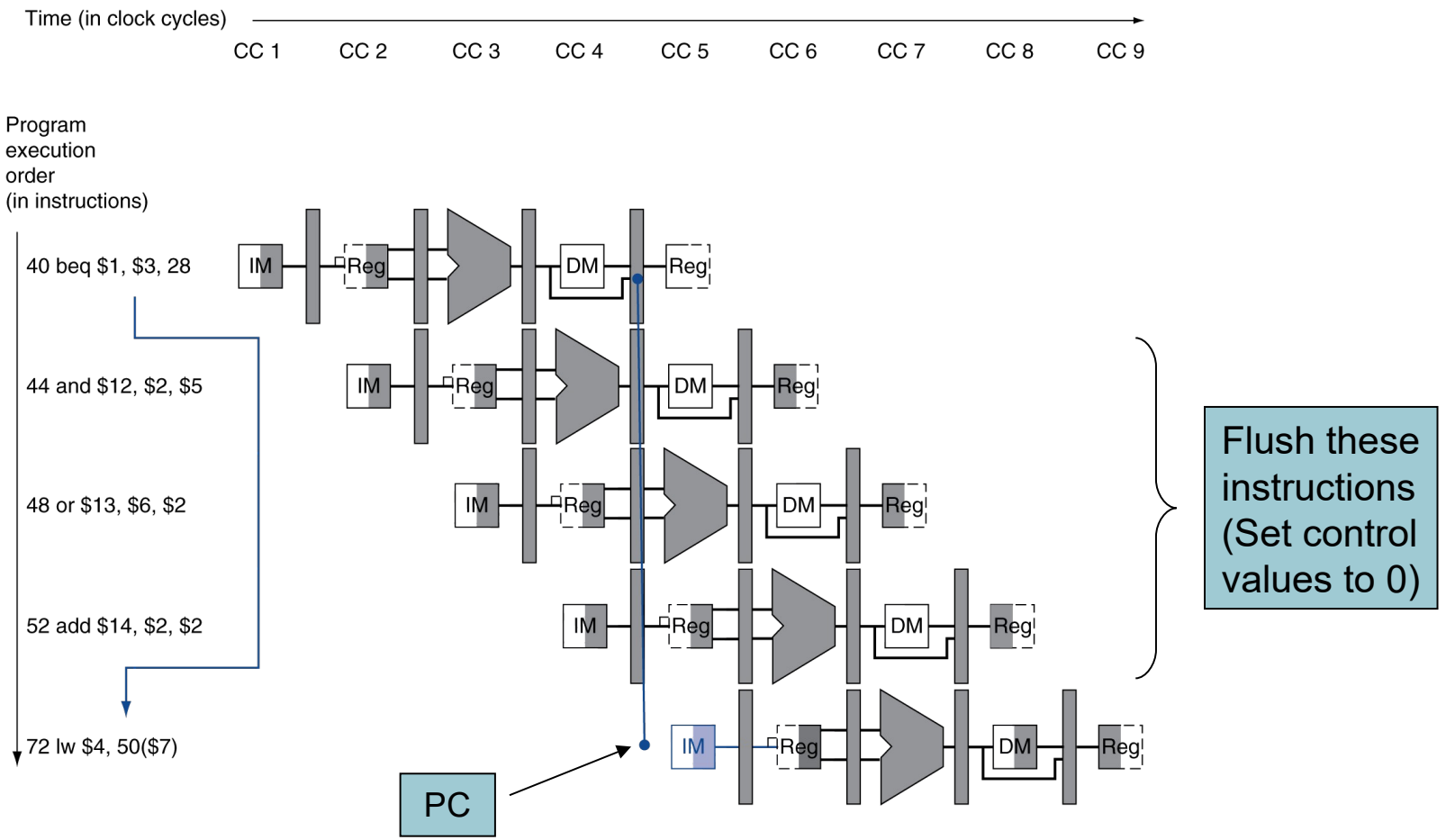
- Hazards involving branches
- **A new instruction must be fetched at every clock cycle to sustain the pipeline**.
- In our design the decision about whether to branch doesn't occur until the MEM pipeline stage comes.
- **This delay in determining the proper instruction to fetch is called a control hazard**

Pipelined Control



Branch Hazards

- If branch outcome determined in MEM



Branch hazards

- They are simple to understand
- They occur less frequently than data hazards
- There is nothing as effective solution against control hazards as **forwarding** is against data hazards
- Two simpler techniques
  - Assume branch not taken (static prediction)
  - Dynamic branch prediction

- A common improvement over branch stalling is to assume that the branch will **not be taken** and thus continue execution down the sequential instruction stream.
- If the branch is **taken** the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.
- Discarding the instructions means, we must be able to flush the instructions in the IF, ID and EX stages of pipe.

Reducing the Delay of Branches

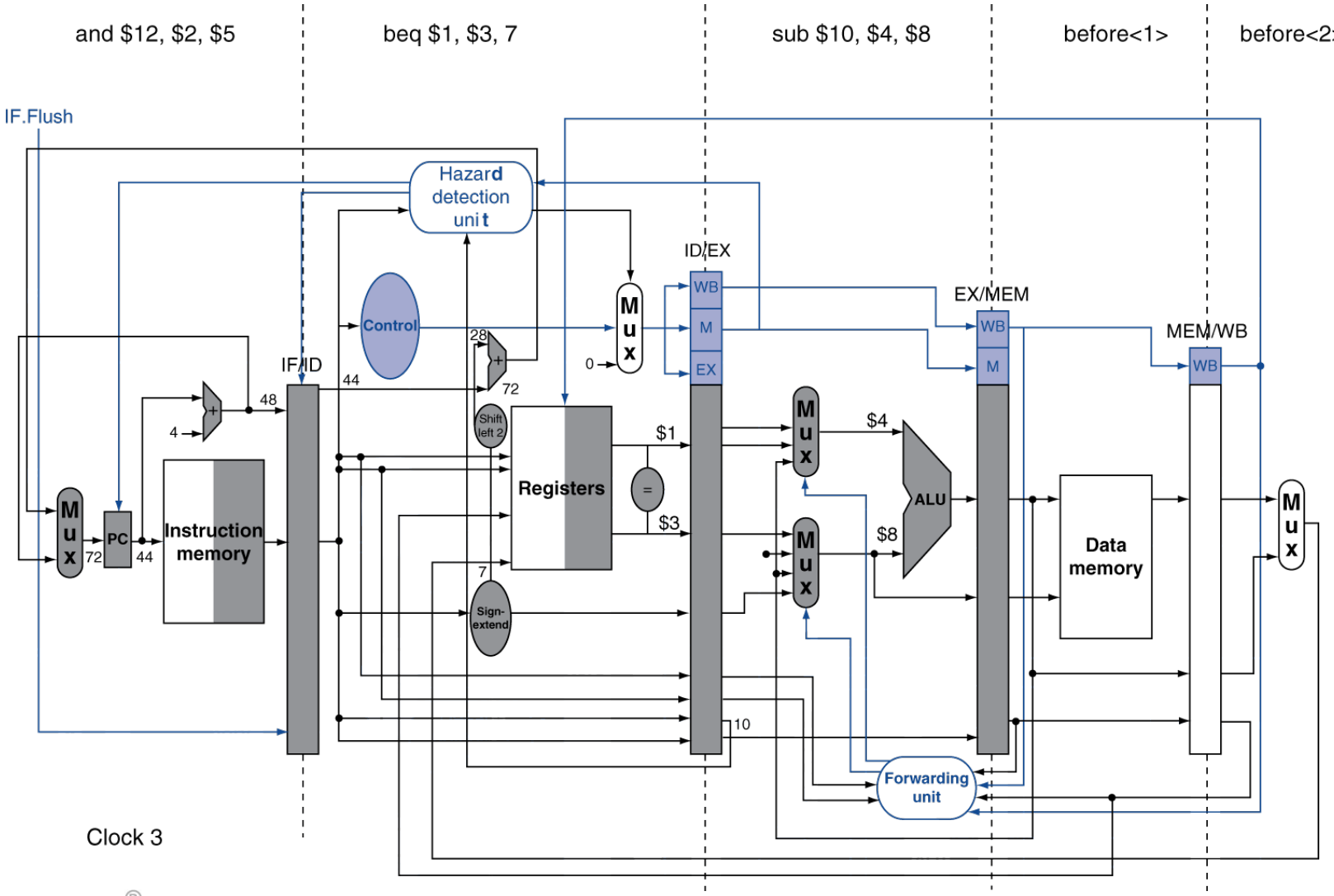
- One way to improve branch performance is to reduce the **cost of the taken branch**
- If we move the branch execution earlier in the pipeline, then fewer instructions need be flushed
- Branches rely only on simple tests
- So move branch decision up into ID stage

Reducing Branch Delay

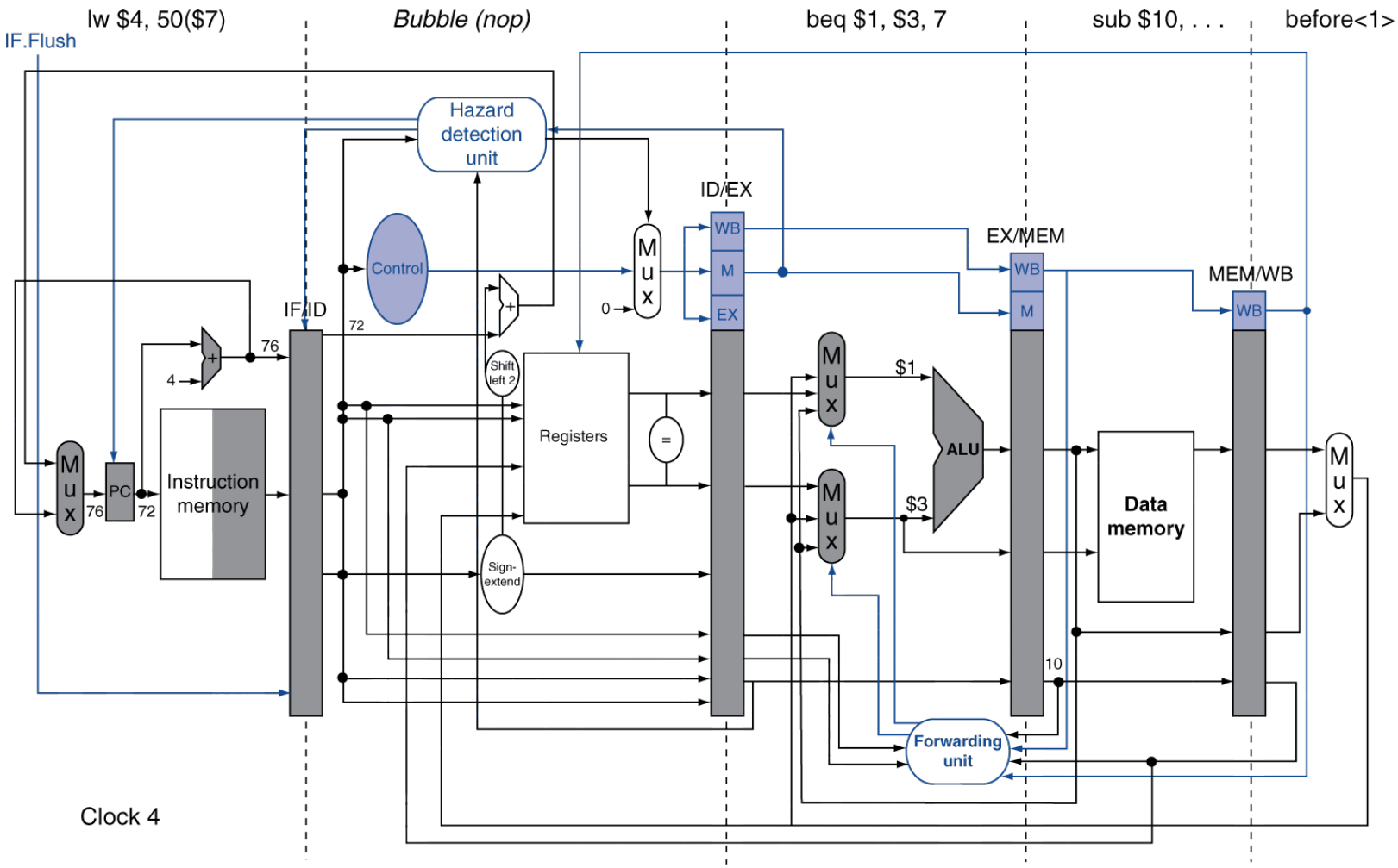
- Moving the branch decision up requires two actions to occur earlier
  - Computing the branch target address
  - Evaluating the branch decision
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7
44:   and   $12, $2, $5
48:   or    $13, $2, $6
52:   add   $14, $4, $2
56:   slt   $15, $6, $7
      ...
72:   lw    $4, 50($7)
```

Example: Branch Taken



Example: Branch Taken



- Harder part is the branch decision itself
- Moving the branch test to the ID stage implies **additional forwarding and hazard detection hardware**, since a branch dependent on a result still in the pipeline
- So we will need to forward results to the equality test logic that operates during ID.

2 complicating factors

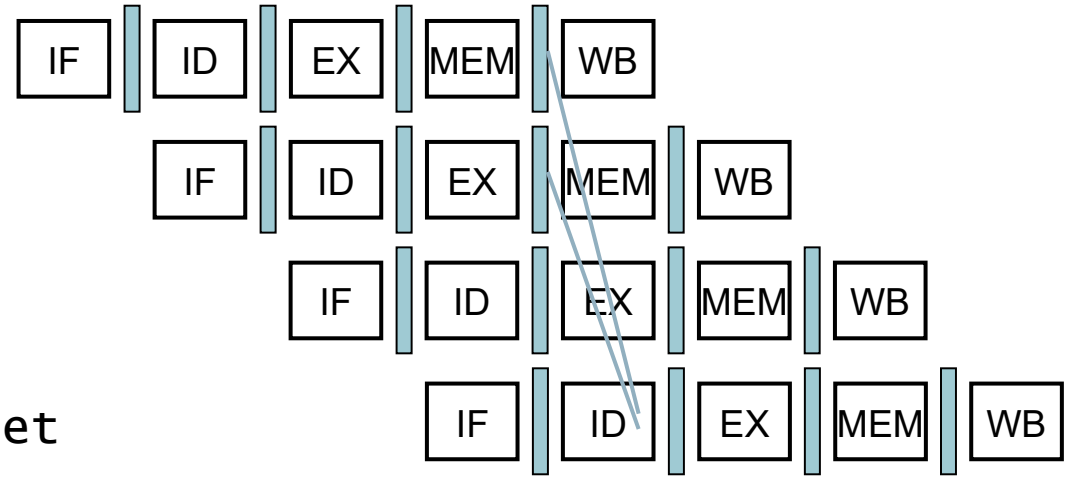
- The introduction of equality test unit in ID will require new **forwarding logic**. The bypassed source operands can from the ALU/MEM or MEM/WB stages.
- The values in a branch comparison are needed during ID but may be produced later in time, **a stall** will be needed.

- **Despite these difficulties**, moving the branch execution to the ID stage is an **improvement**
- Because it reduces the penalty of a branch to only one instruction if the branch is taken(The one currently being fetched)

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

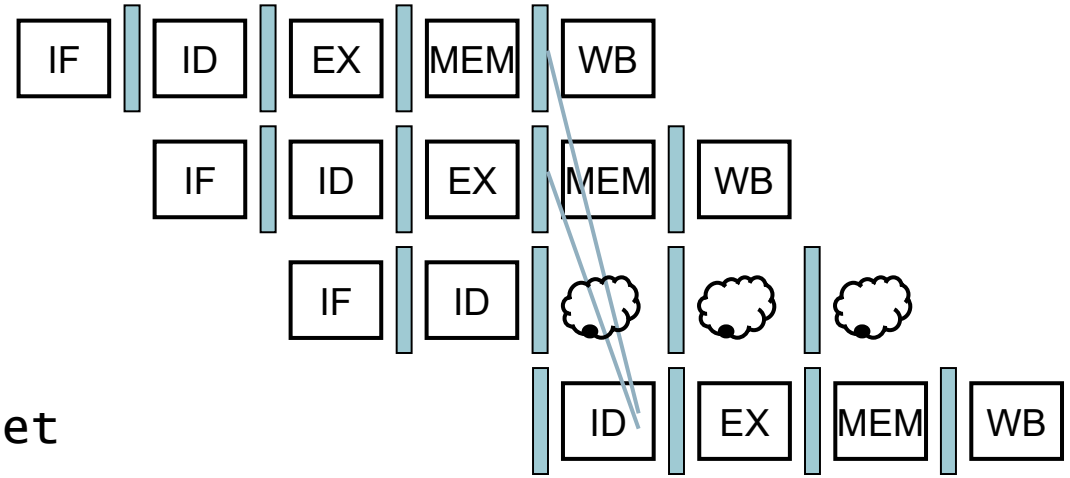


- Can resolve using forwarding

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

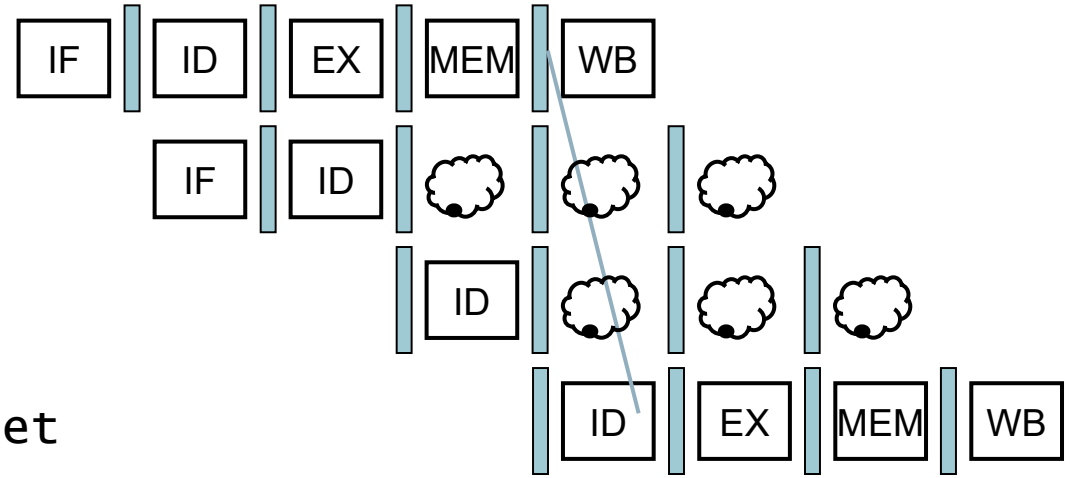
```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```



- To flush instructions in IF stage, we add a control line called IF.Flush that zeros the instruction field of IF/ID pipeline register
- Clearing the register transforms the fetched instruction into a *nop*

Dynamic branch prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- **With more hardware** it is possible to try to predict branch behavior **during program execution time**
- One approach is to lookup the address of the instruction to **see if a branch was taken the last time this instruction was executed**

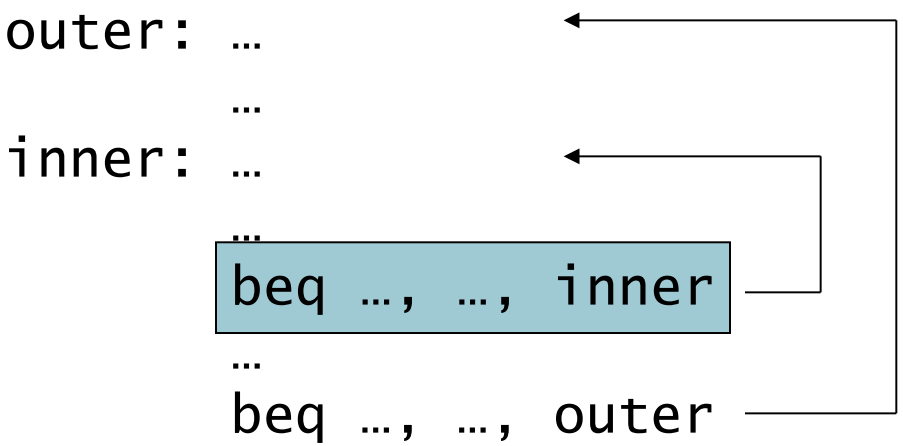
- One implementation of that approach is **branch history table**(BHT)
- BHT is a small memory indexed by the lowest portion of the address of the branch instruction
- The memory contains a bit that says whether the branch was recently taken or not

- Starts off as T, flips when ever the branch behaves opposite to prediction

- Limitation: even if a branch is almost always taken, we will predict incorrectly twice, rather than once, when it is not taken

1-Bit Predictor: Shortcoming

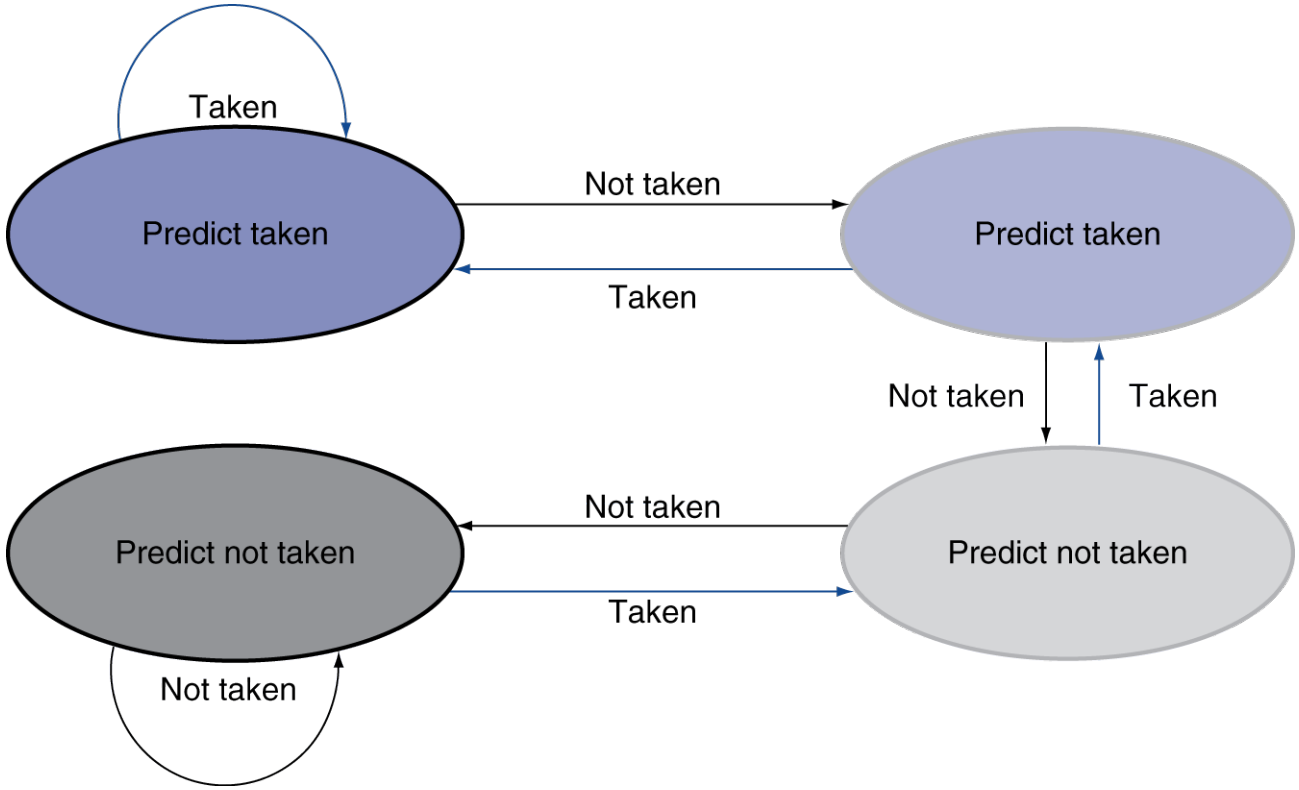
- Inner loop branches mispredicted **twice**!



  - **Mispredict as taken on last iteration of inner loop**
  - **Then mispredict as not taken on first iteration of inner loop next time around**

2-Bit Predictor

- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately