ELT3047 Computer Architecture

Lecture 10: Pipelined Processor Design (cont.)

Hoang Gia Hung
Faculty of Electronics and Telecommunications
University of Engineering and Technology, VNU Hanoi

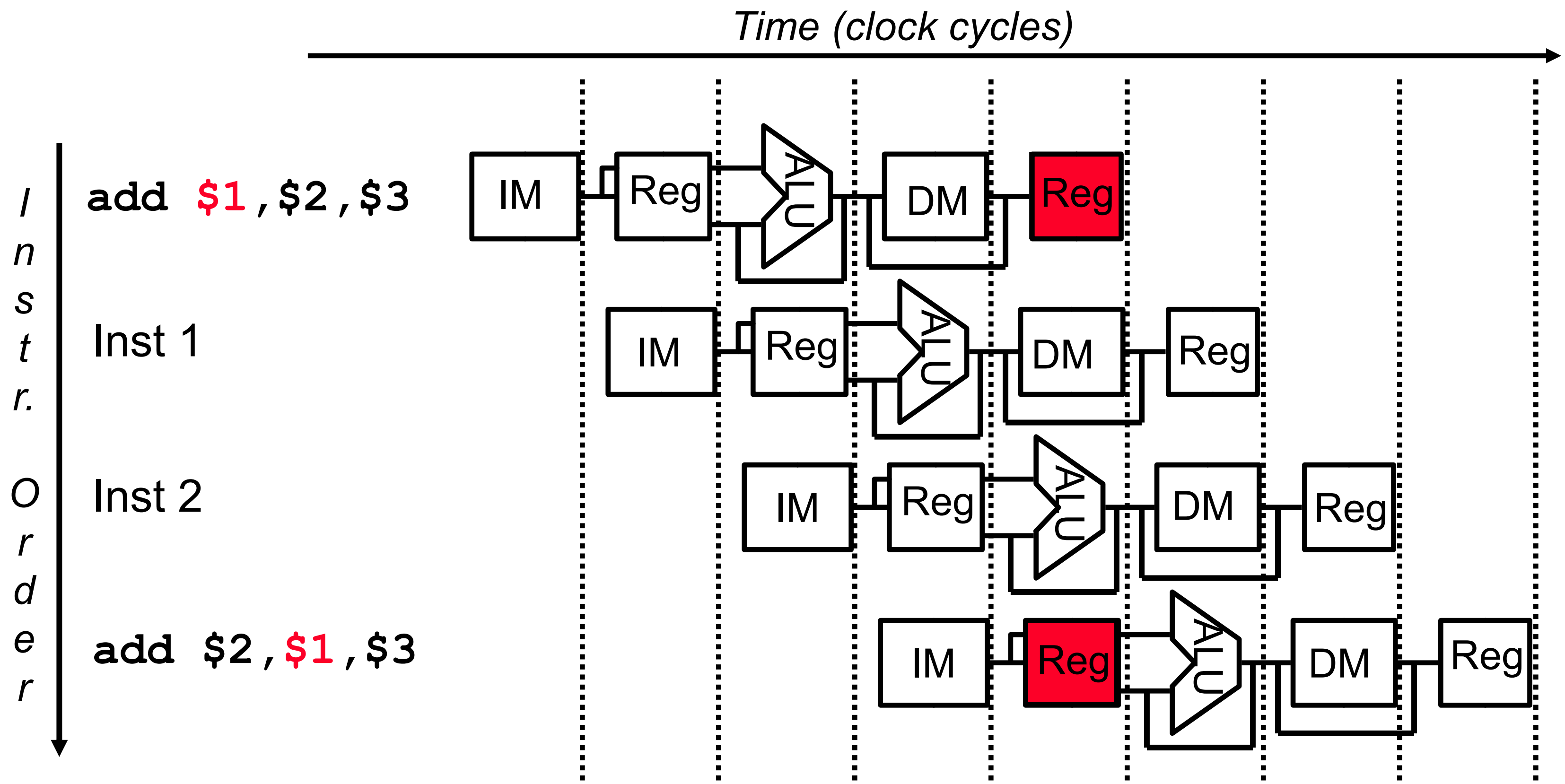
Last lecture review

- Multi-cycle processor
  - Use **one clock cycle per step** → shorter clock cycle time
  - Higher performance over single-cycle processor due to less waste
- Pipeline processor design
  - Employs **instruction parallelism**: process the next instruction on the resources available when current instructions move to subsequent phases.
  - Speedup is due to **increased throughput**: once the pipeline is full, CPI=1.
  - Datapath is derived from single-cycle case with additional buffer registers
  - Some control signals are moved along the pipeline via inter-stage buffers.
- As the instruction pipeline is not ideal, various issues may occur including structural, data, and control **hazards**.
- **Today's lecture**: handling of pipeline hazards

Pipeline hazards

- Issues in pipeline design
  - **structural hazards**: attempt to use the same resource by two different instructions at the same time
  - **data hazards**: attempt to use data before it is ready, e.g. an instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - **control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated (e.g. branch and jump instructions, exceptions)
- Serious problems, cannot be ignored
- **Design objectives**: keeping the pipeline **correct**, **moving**, and **full** in the presence of events that disrupt pipeline flow.

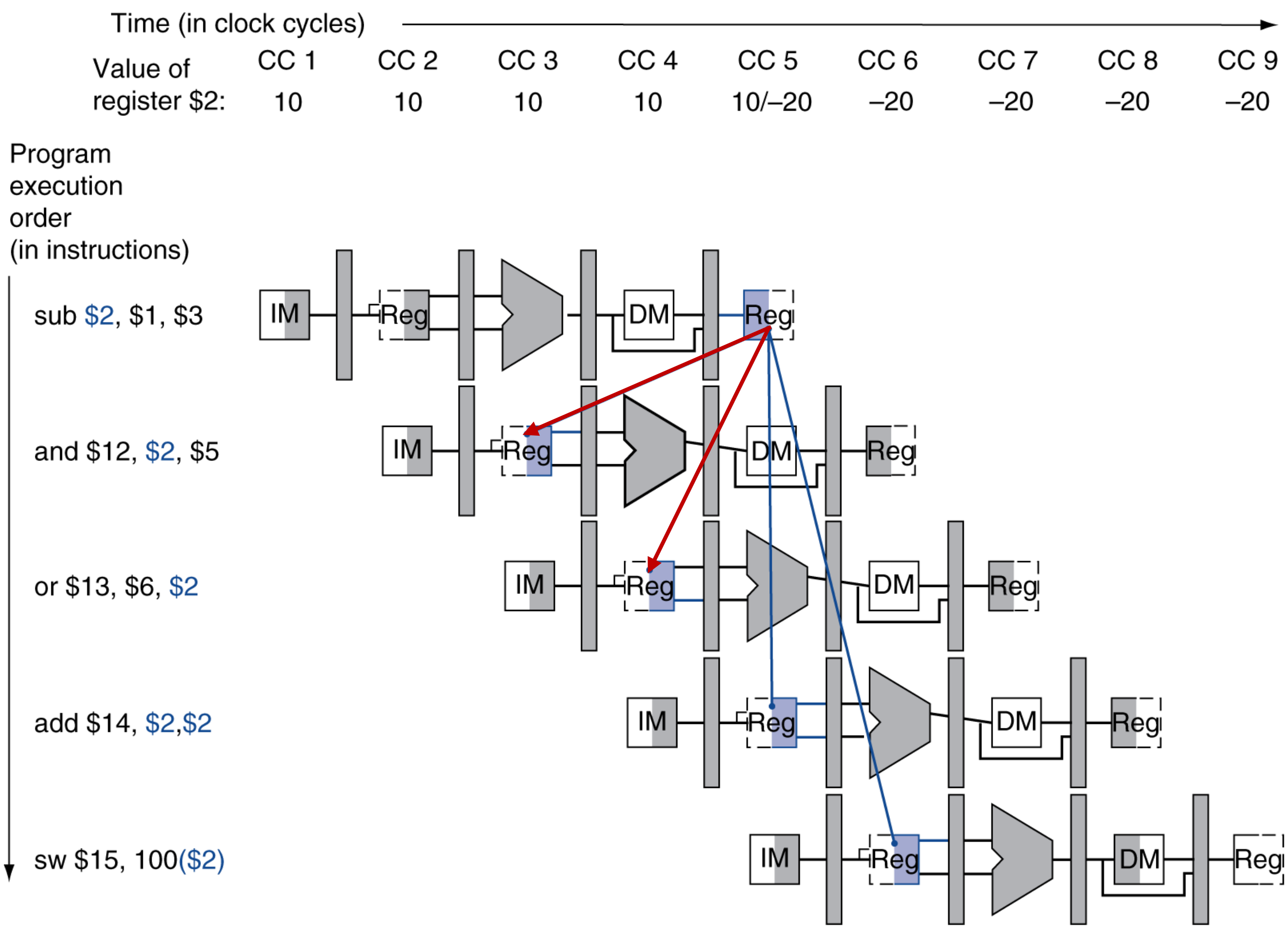
Structural hazard: example



- Two instructions are attempting to use the same register ($1) during the same cycle (CC5).

Data hazard: example

- Dependencies backward in time: **read before write** is ready

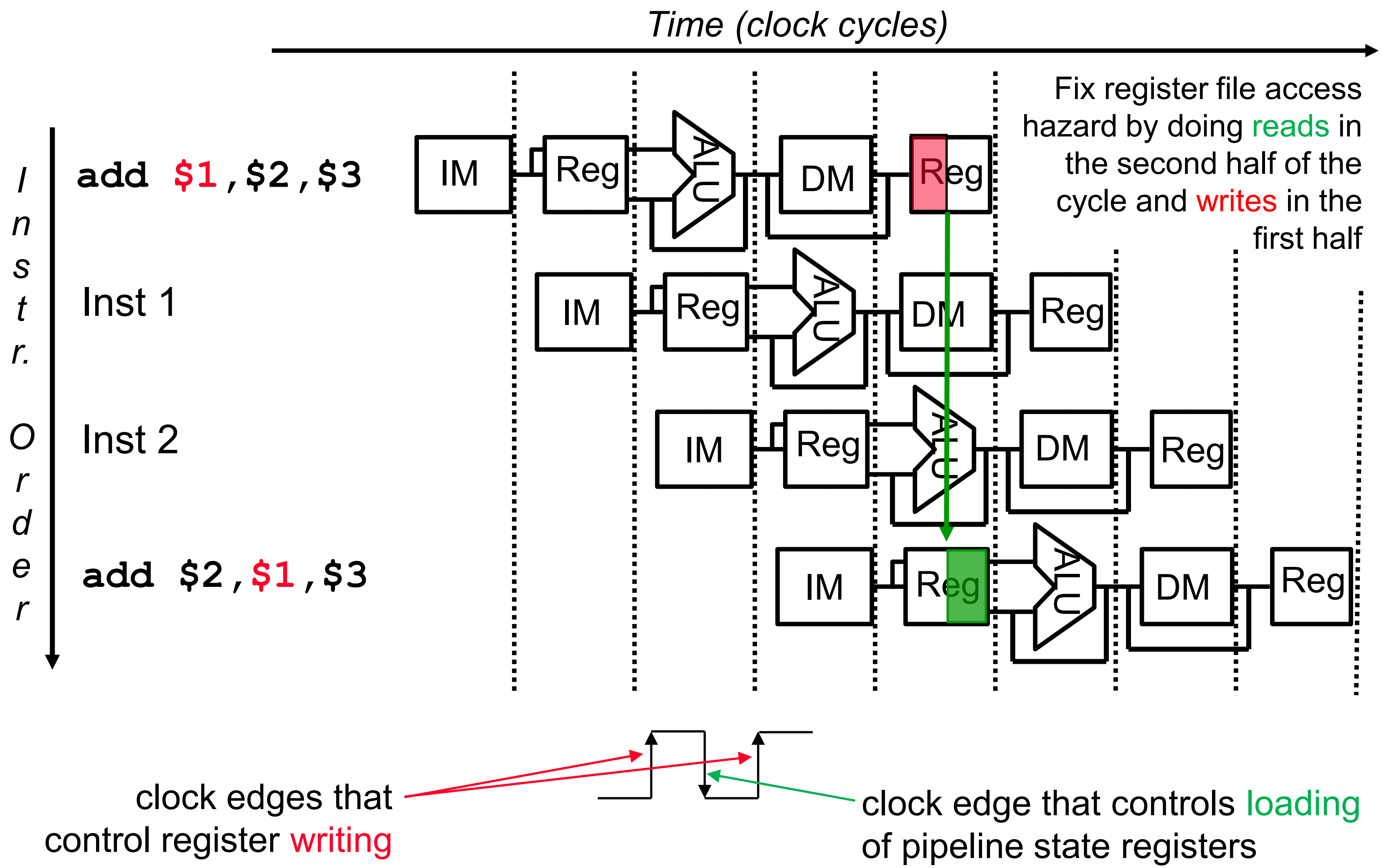


Hazard handling methods

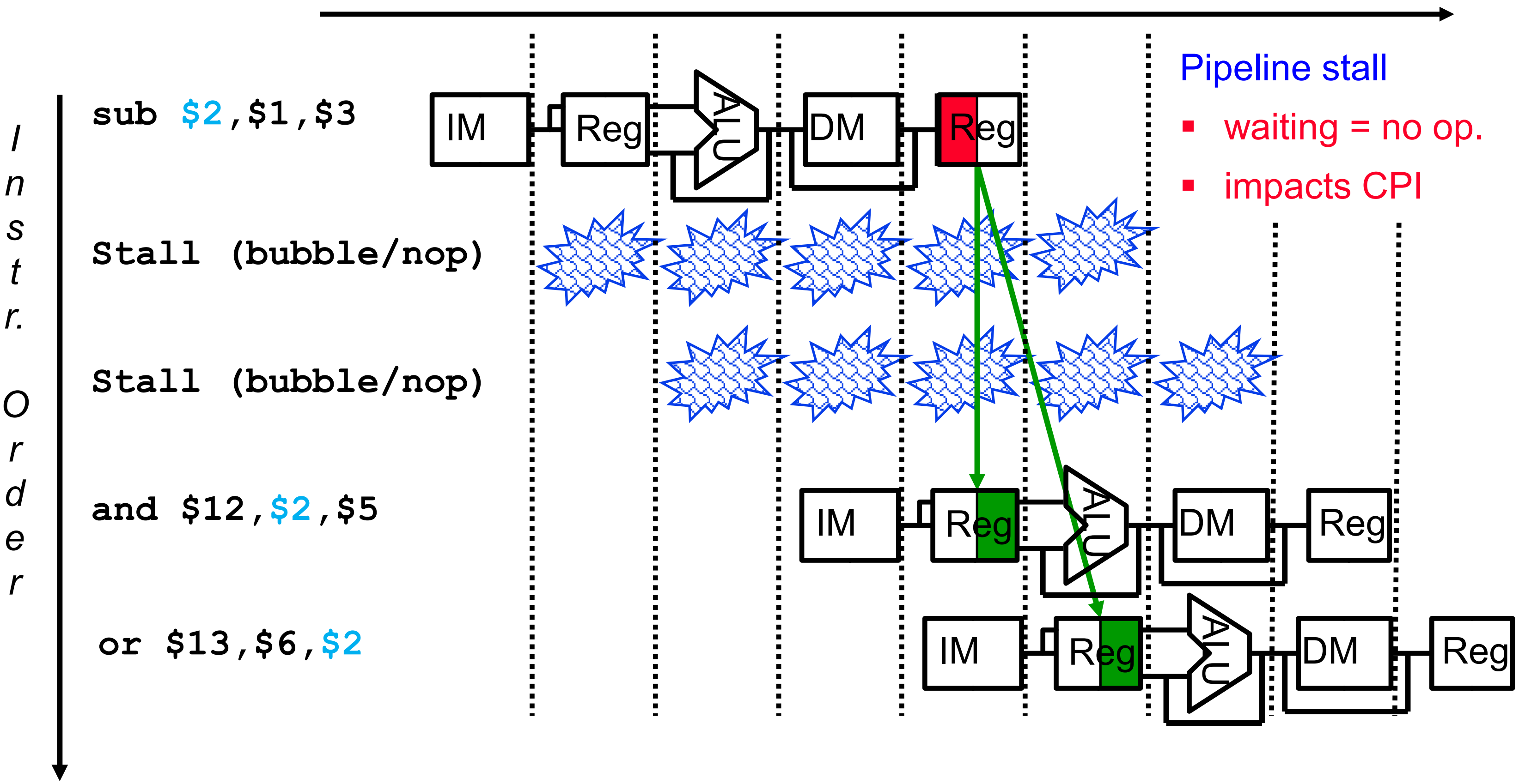
- General ways of handling **structural** hazard
  1. **Stall:** delay access to resource
     - e.g., detect and wait until value is available in register file
  2. **Add more hardware resources**: increase the throughput
     - more costly, e.g. use separate memories for instructions & data
- Five fundamental ways of handling **true data hazard**
  1. **Stall**: detect and wait
  2. **Forward**: detect and forward/bypass data to dependent instruction
  3. **Eliminate**: detect and eliminate the dependence at the **software** level
     - No need for the hardware to detect dependence
  4. **Predict**: predict the needed value(s), execute "speculatively", **and verify**
  5. Do something **else** (fine-grained multi-threading)
     - No need to detect

- **Stall** can resolve any type of hazards (data/control/structural)

Structural hazard handling example



Data hazard handling: pipeline stall



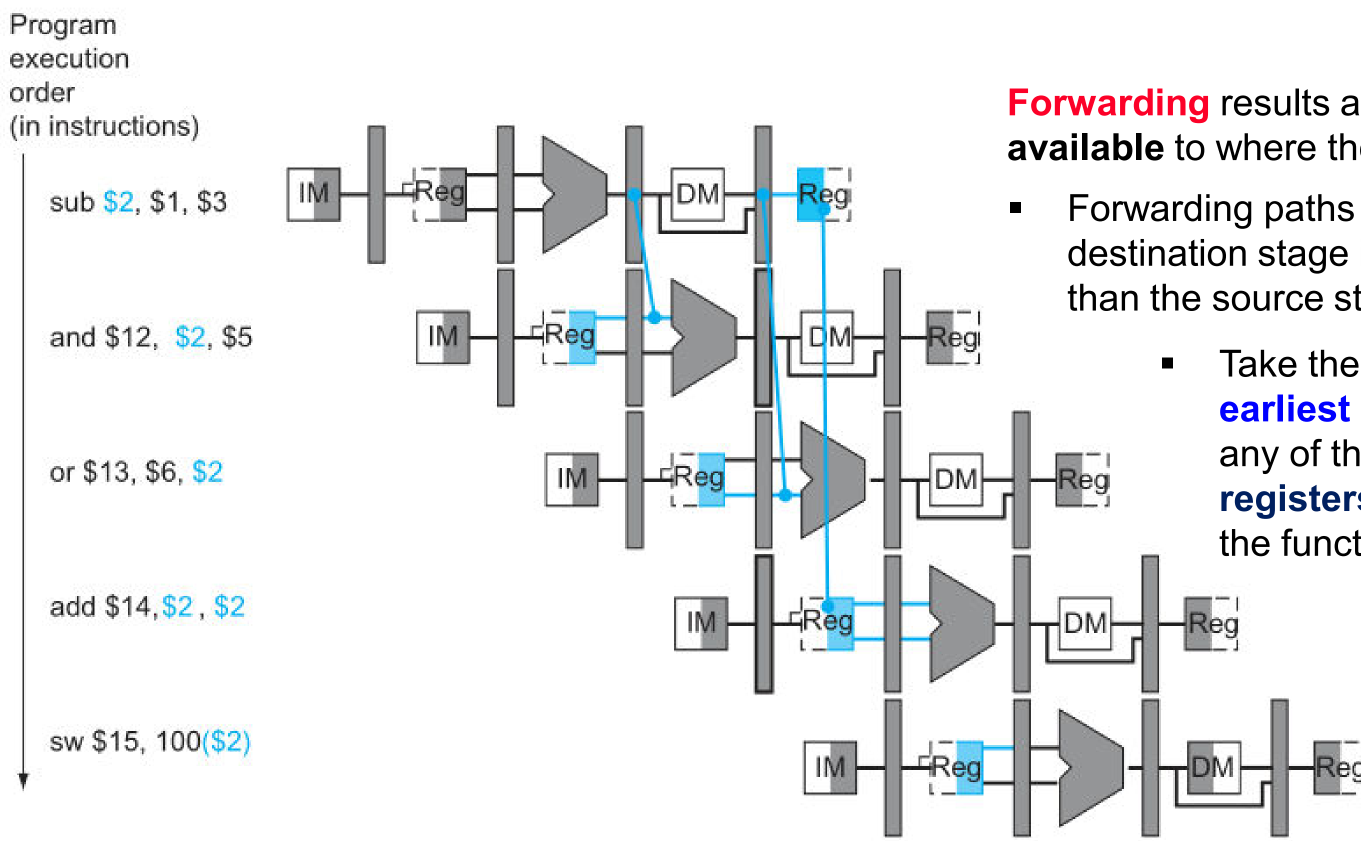
Pipeline stall

- waiting = no op.
- impacts CPI

Data Forwarding/Bypassing: overview

Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X

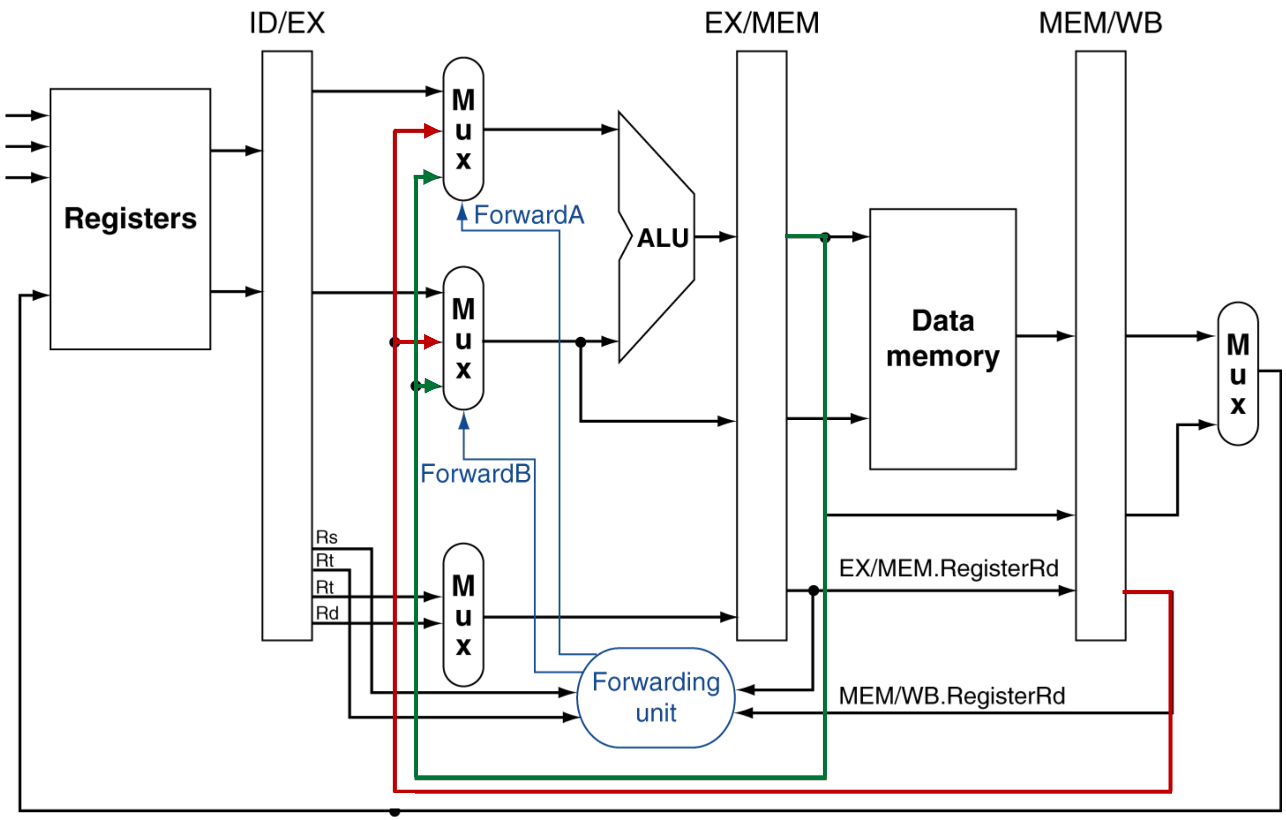


Forwarding results as soon as they are **available** to where they are **needed**.

- Forwarding paths are valid only if the destination stage is **later** in time than the source stage.
    - Take the result from the **earliest** point that exists in any of the pipeline **state registers** and forward it to the functional unit (**ALU**).

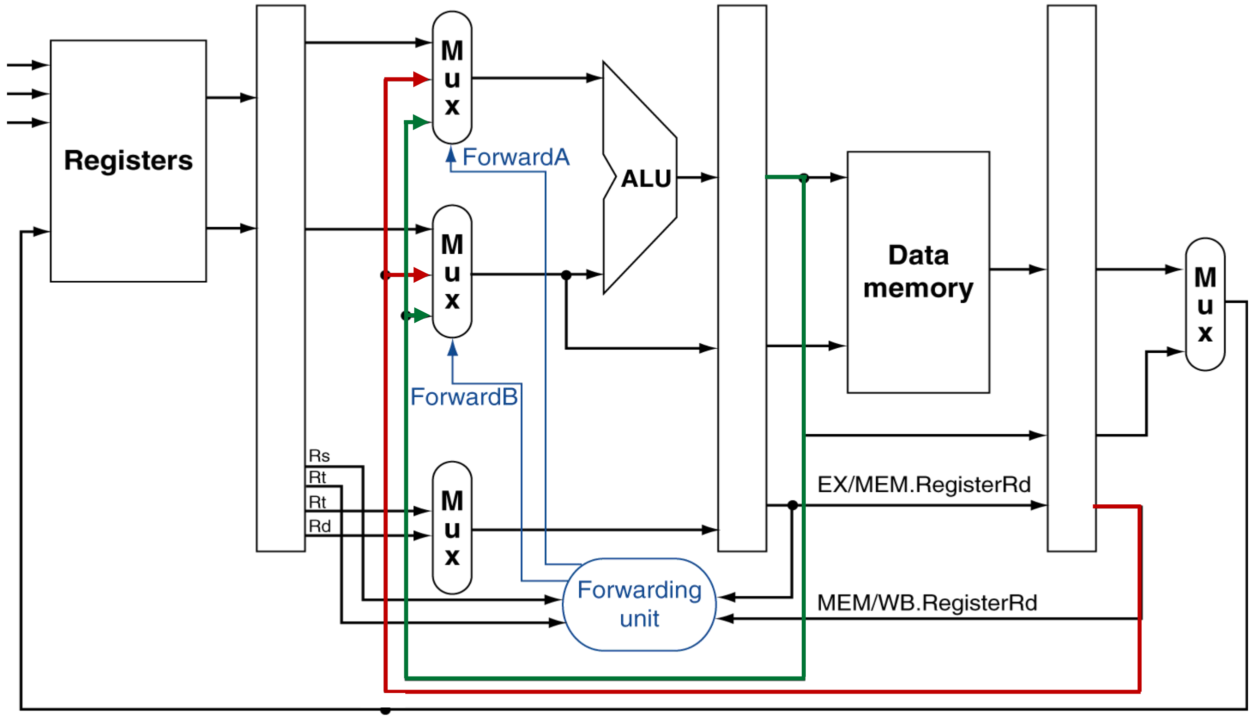
Forwarding: implementation

- Data from **EX/MEM**, **MEM/WB stage** pipeline registers & is fed back to two multiplexers at the inputs of the ID/EX stage.
  - Add a **Forwarding unit** to calculate 2 control signals **ForwardA**&**B**.



b. With forwarding

Forwarding: design of control signals



Mux control	Source	Explanation
Forward**A** = **00**	ID/EX	The **first** ALU operand comes from the **register** file.
Forward**A** = **10**	EX/MEM	The **first** ALU operand is forwarded from the **prior ALU** result.
Forward**A** = **01**	MEM/WB	The **first** ALU operand is forwarded from **data memory** or an earlier ALU result.
Forward**B** = **00**	**ID/EX**	The **second** ALU operand comes from the register file.
Forward**B** = 10	**EX/MEM**	The **second** ALU operand is forwarded from the prior ALU result.
Forward**B** = 01	**MEM/WB**	The **second** ALU operand is forwarded from data memory or an earlier ALU result.

Forwarding conditions

- Pipelined datapath convention:
  - Register numbers are passed along the pipeline, e.g. EX/MEM.RegisterRd = register number for Rd sitting in EX/MEM pipeline register.
  - ALU operands in EX stage: ID/EX.RegisterRs, ID/EX.RegisterRt.
- Data hazards when
  1. EX/MEM.RegisterRd = ID/EX.RegisterRs
  2. EX/MEM.RegisterRd = ID/EX.RegisterRt
  3. MEM/WB.RegisterRd = ID/EX.RegisterRs
  4. MEM/WB.RegisterRd = ID/EX.RegisterRt

  Fwd from EX/MEM pipeline register

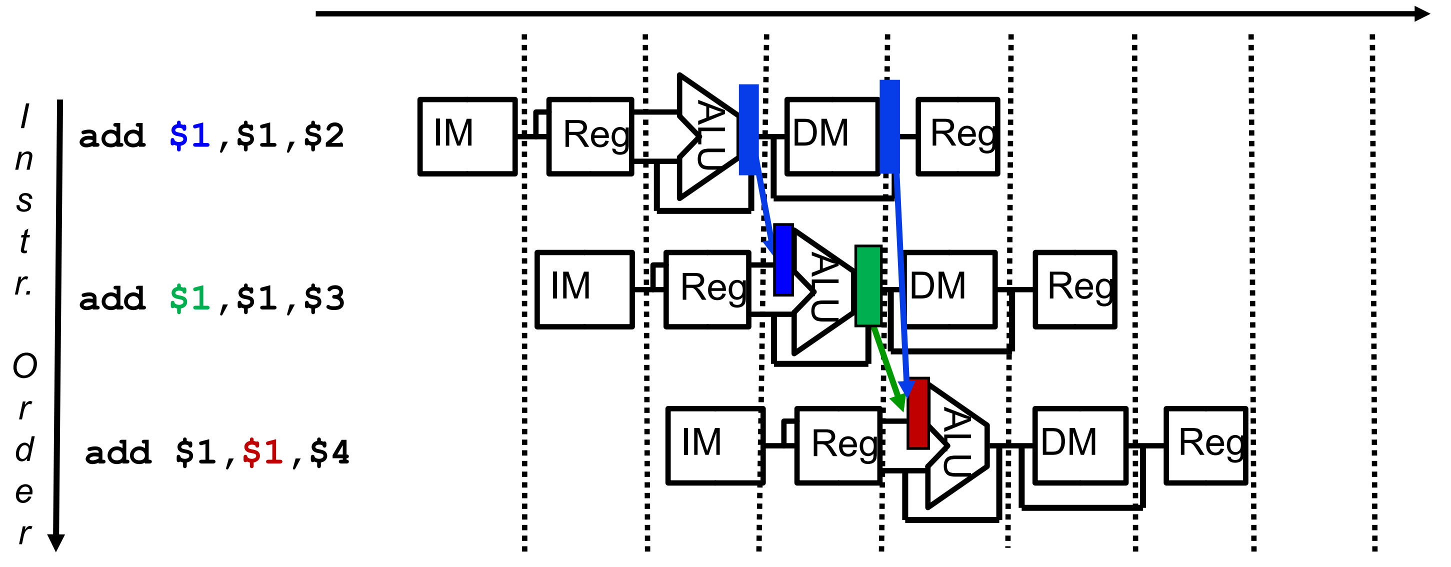
  Fwd from MEM/WB pipeline register
- But only if forwarding instruction will write to a register!
  - Avoid forwarding when it shouldn't: check if EX/MEM.RegWrite=1 (e.g. `add`), MEM/WB.RegWrite=1 (e.g. `lw`)
- And only if $Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

Forwarding: control algorithm

- EX hazard:
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard:
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Forwarding: yet another complication

- Double data hazard



- There is a conflict between the result of the EX stage instruction and the MEM stage instruction → which should be forwarded?
  - The more recent result (EX stage) should be forwarded.
- Revise forward conditions for MEM hazard
  - Only forward if EX hazard condition isn't true

Forwarding: revised control algorithm

- Revised control for MEM hazard (with the additions highlighted)
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)

    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)

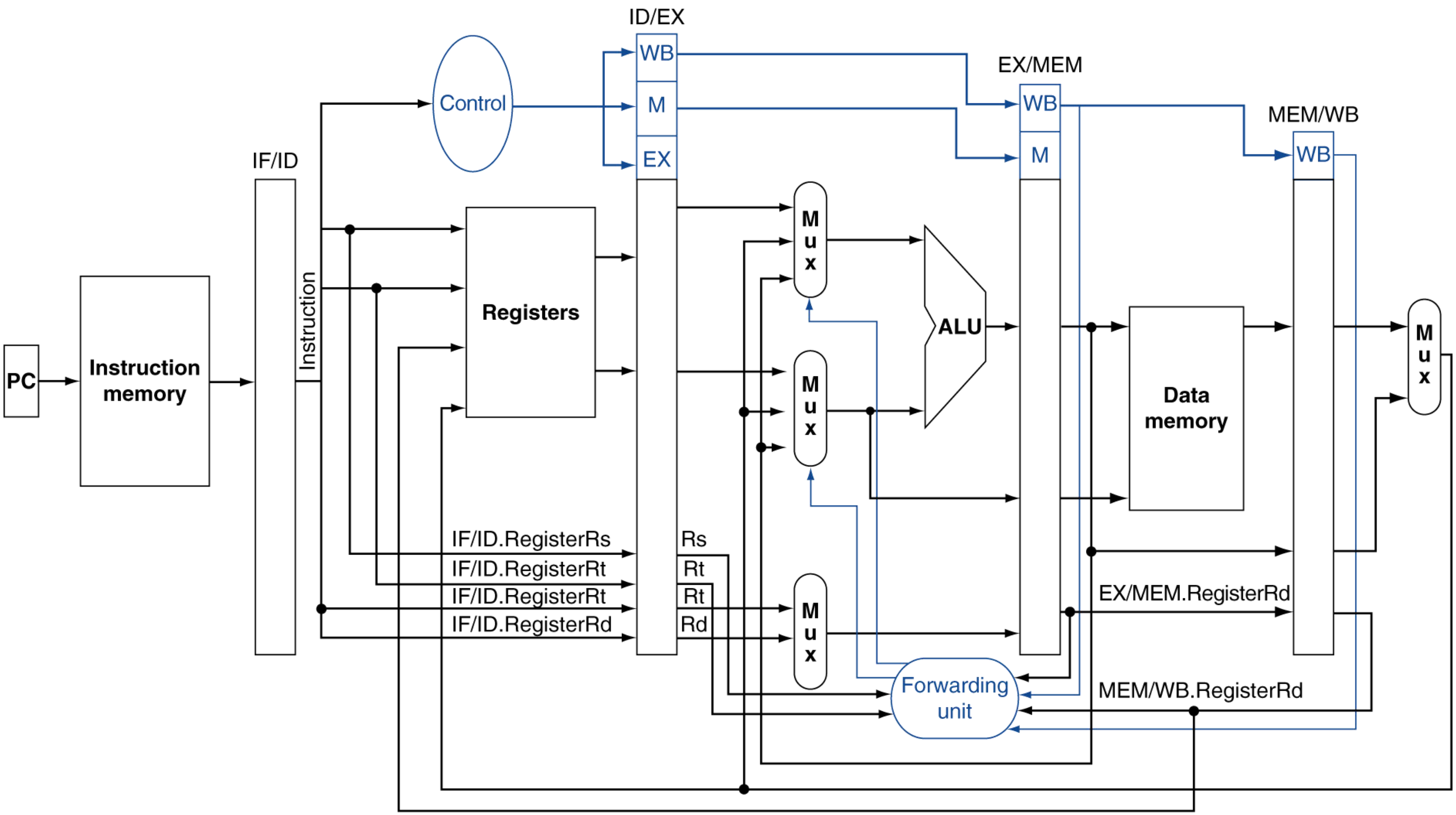
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))

    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))

    ForwardB = 01

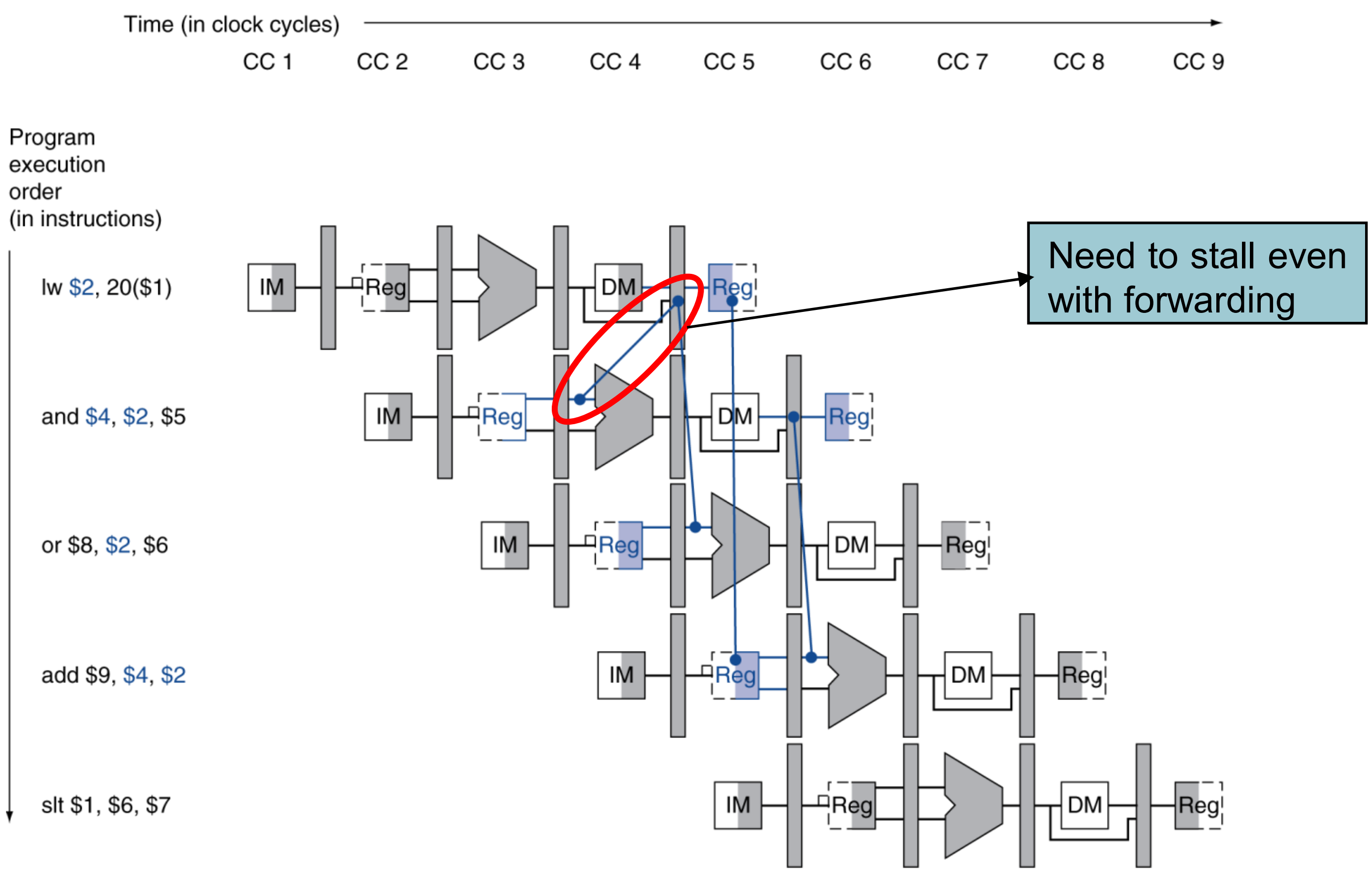
Datapath with forwarding



- Does forwarding solve all our problems?

Load-Use Data Hazard

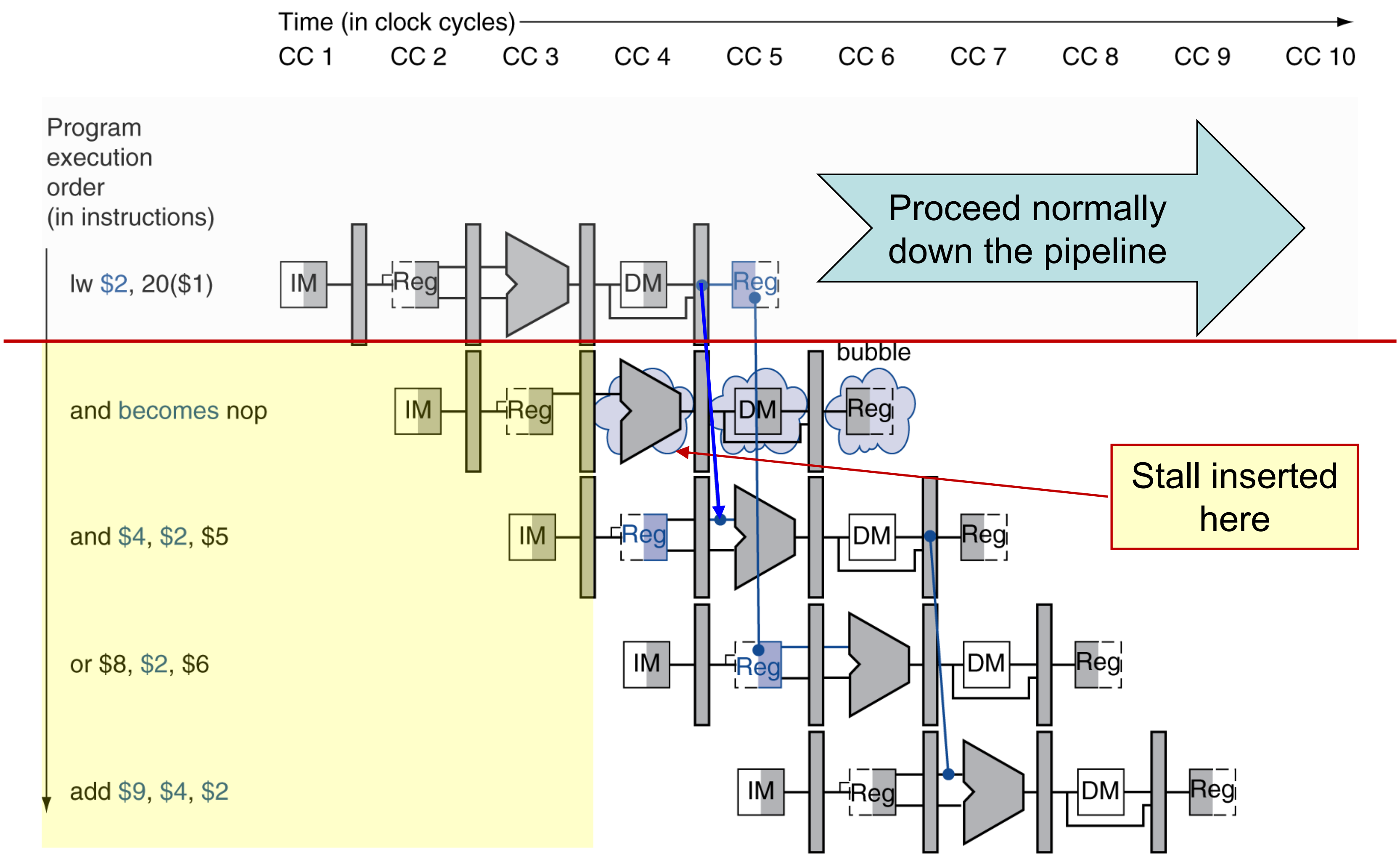
- Unfortunately, not all data hazards can be forwarded
  - **Load** has a delay that **cannot** be eliminated by forwarding



Load-Use Hazard Detection

- Read-after-Write (RAW) hazard after a load
  - The `load` instruction will be in the **EX** stage while the using instruction (that depends on the load data, e.g., `and`) is in the **ID** stage
- Condition for stalling the pipeline
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected (when `lw` is in EX stage), insert a **bubble** in between `lw` and the dependent instruction the execution stream.
  - A bubble = a **nop** that wastes one clock cycle → all instructions beginning with the using instruction (`and`) are delayed one cycle.
    - `lw` & the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline.
    - After this stall, using instruction (`and`) is decoded again while the following instruction (`or`) is fetched again.
  - Stall allows MEM to read data for `lw` → can now forward to EX stage.

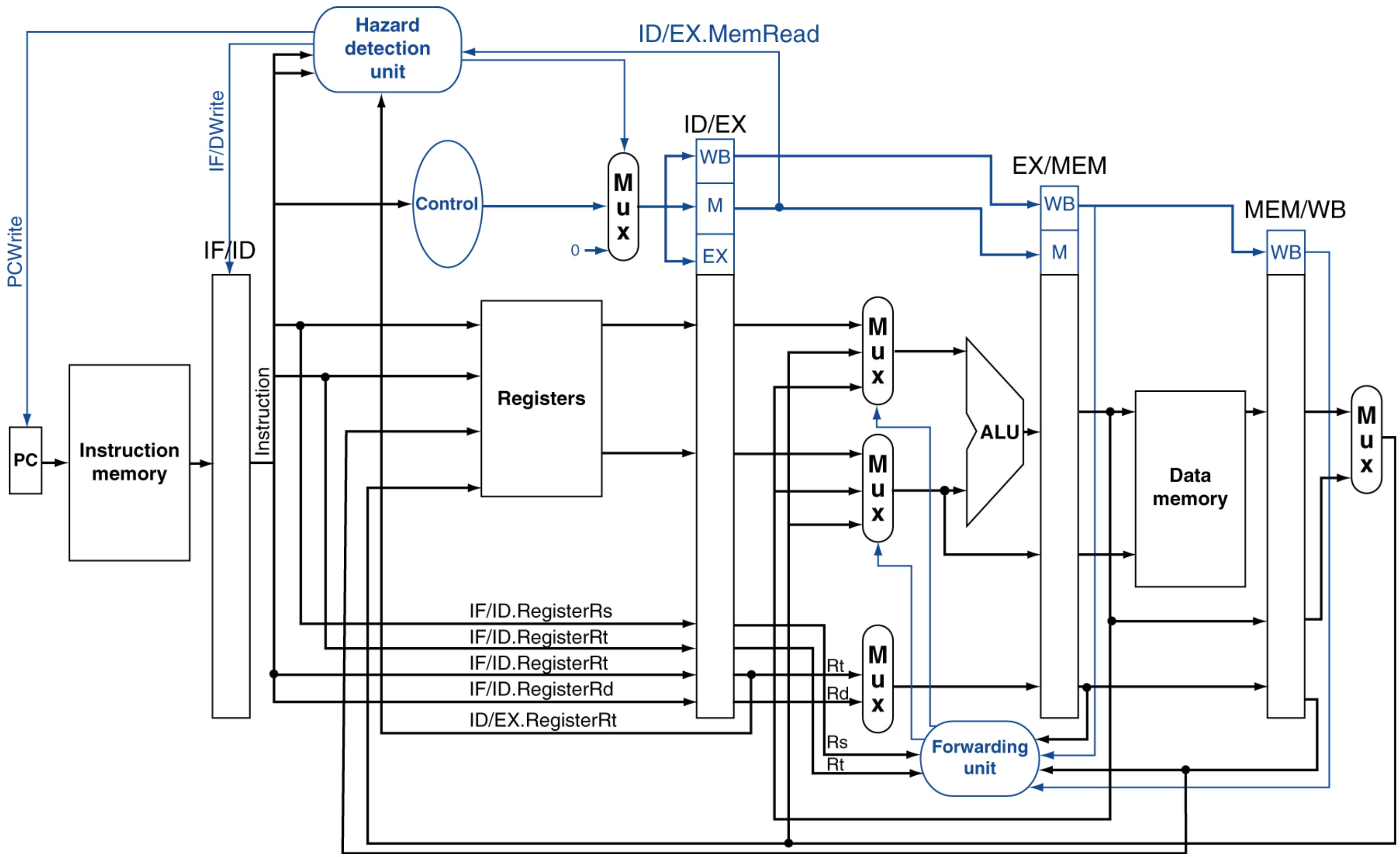
Load-Use Hazard: stalling & forwarding



Stall Hardware

- Prevent instructions in the IF and ID stages from progressing down the pipeline
  - Done by preventing the PC & the IF/ID pipeline registers from changing and deasserting EX, MEM, and WB control fields of the ID/EX pipeline register.
  - These control values are percolated forward at each clock cycle with the proper effect: no registers or memories are written if they are all 0.
- Need a **hazard detection unit**
  - to detect the case & implement the stall by:
  - controlling the writing of PC and IF/ID registers
    - control signals: PCWrite, IF/IDWrite
  - controlling a multiplexor that chooses between the real control values and all 0s.

Datapath with Hazard Detection



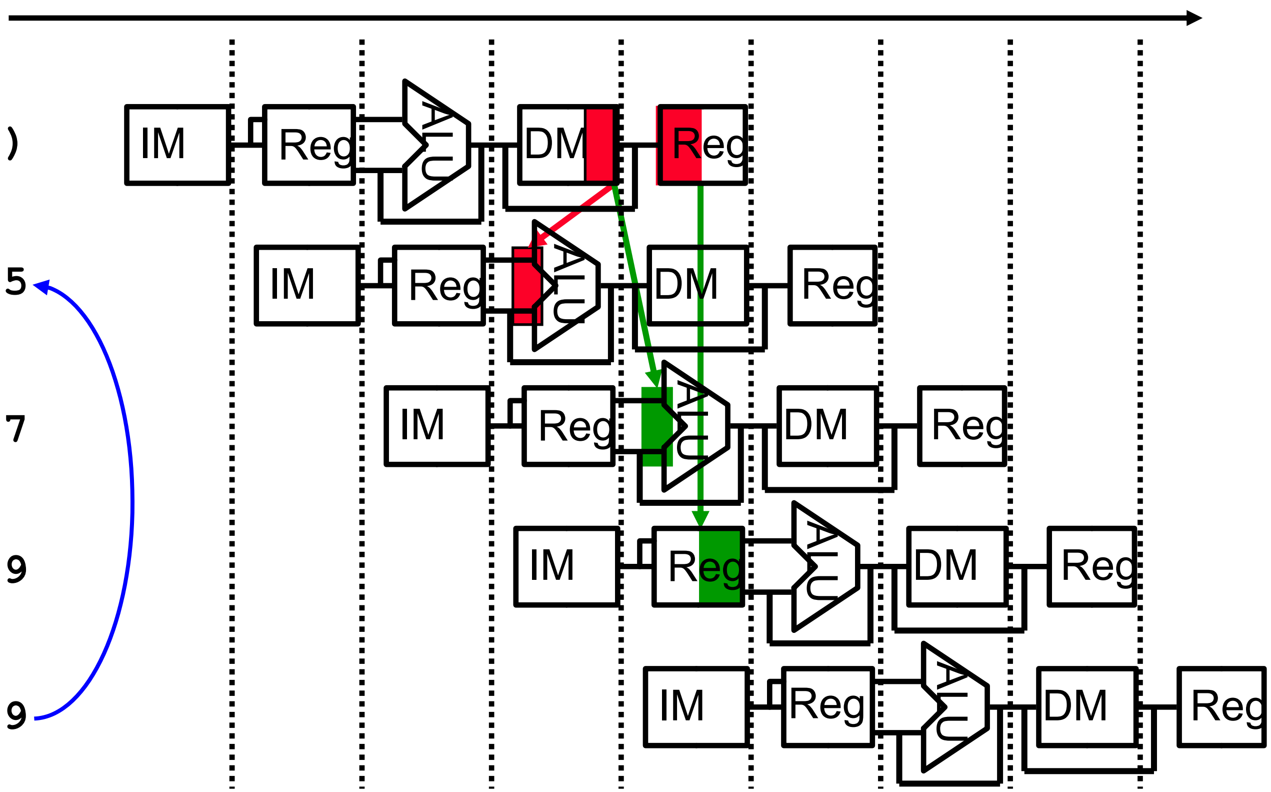
Hazard elimination: compiler scheduling

- Compiler rearranges instructions to eliminate load-use hazard!
  - Also called **static scheduling** <> **dynamic scheduling** (hardware can execute instructions **out of the compiler-specified order**)
  - Requires knowledge of the pipeline structure
- Proebsting & Fischer (1991) show how to optimally schedule a straight line sequence of instructions, given sufficient registers and a delay of one pipeline stage.
  - Build a dependence graph that describes the partial order of instruction definitions and uses
  - Schedule R independent loads (load; load; load; ..)
    - Each load requires a register → R is the minimum number of live registers.
    - Schedule operation independent of the previous load and another load in a pair (operation; load)

Hazard elimination: compiler scheduling example

Instr. Order

```
lw  $1,4($2)
sub $4,$1,$5
and $6,$1,$7
or  $8,$1,$9
xor $2,$7,$9
```

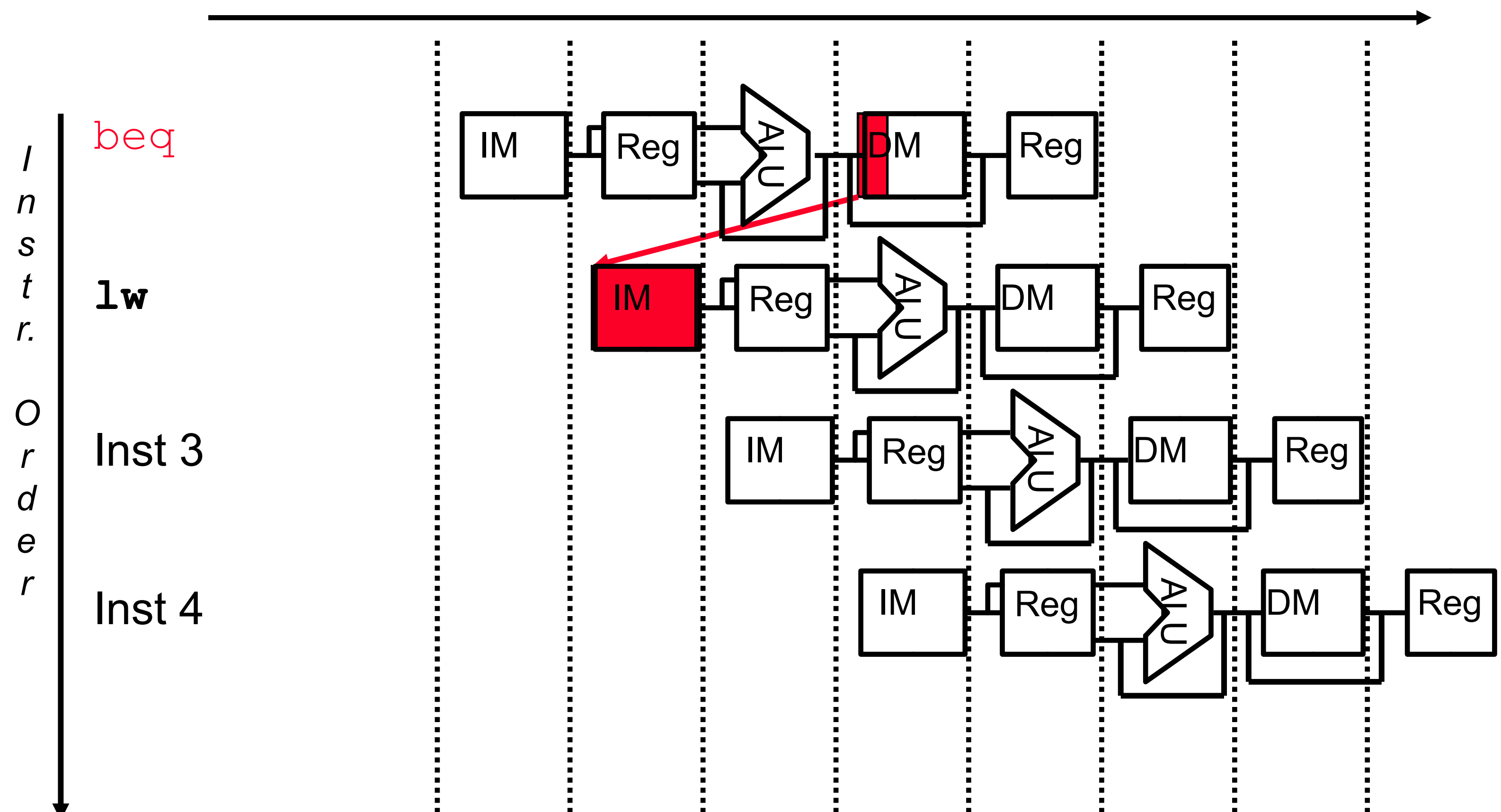


Control Hazards

- When the flow of instruction addresses is **not** sequential (i.e., PC = PC + 4); incurred by change of flow instructions
  - Unconditional branches (`j, jal, jr`)
  - Conditional branches (`beq`, `bne`)
  - Exceptions
- Possible resolution approaches
  - **Stall** (impacts CPI)
  - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Delay decision (requires compiler support)
  - **Predict** and hope for the best!
- Control hazards occur less frequently than data hazards, but there is **nothing as effective** against control hazards as forwarding is for data hazards.

Branch hazards: overview

- Dependencies backward in time cause hazards



Fix a Branch Control Hazard by stalling

Instr. Order

beq

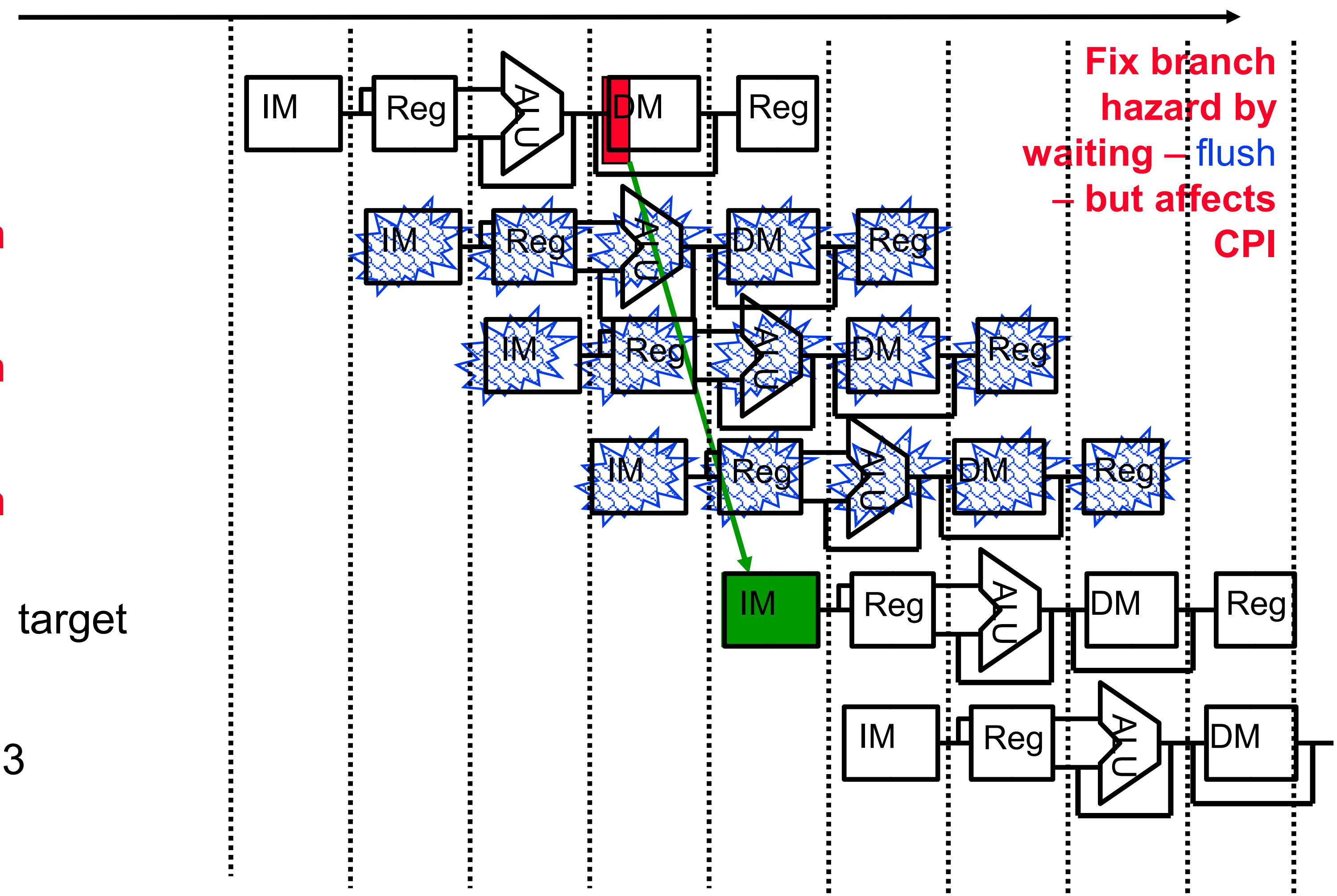
flush

flush

flush

beq target

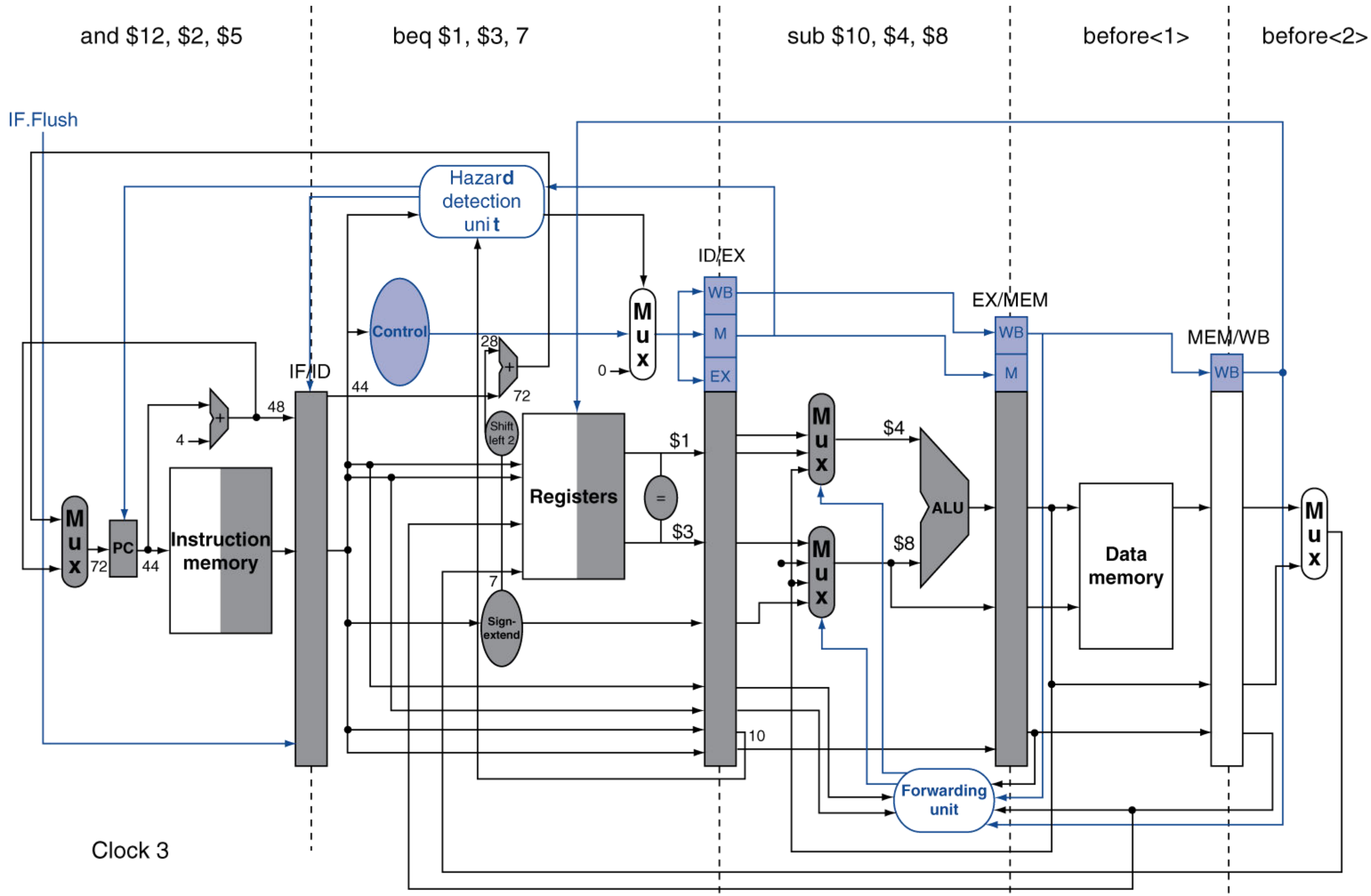
Inst 3



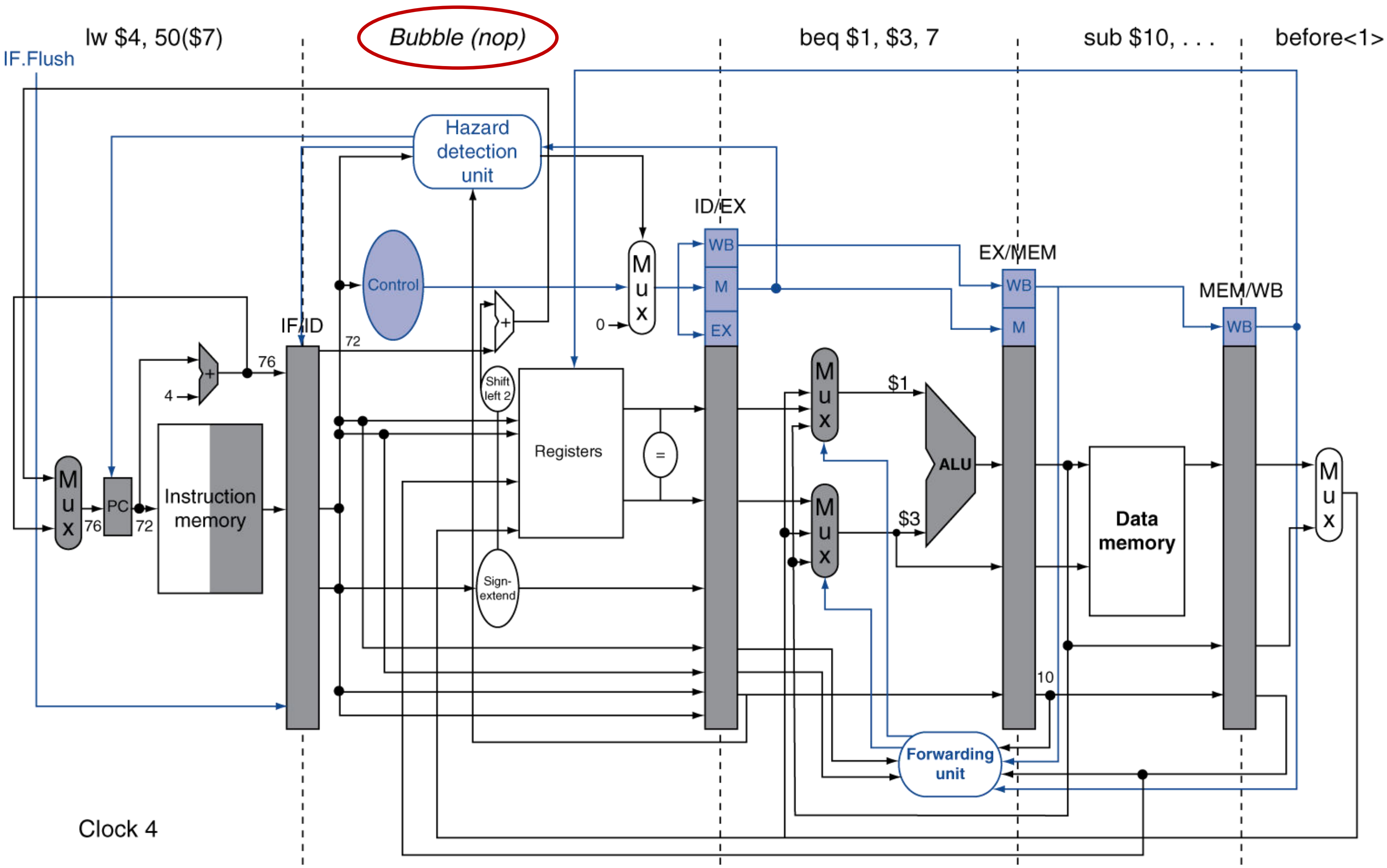
Two “Types” of Stalls

- **nop** instruction (or bubble) is **inserted** between two instructions in the pipeline (c.f., load-use hazards)
  - Keep the instructions earlier in the pipeline (later in the code) from progressing down the pipeline for a cycle **(“bounce” them in place** with write control signals)
  - Insert nop instruction by zeroing control bits in the pipeline register at the appropriate stage
  - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline
- Flushes (or instruction **squashing**) where an instruction in the pipeline is **replaced** with a nop instruction
  - Used for instructions located sequentially after `j` and `beq`
  - Zero the control bits for the instruction to be flushed

Example: pipline before flushing



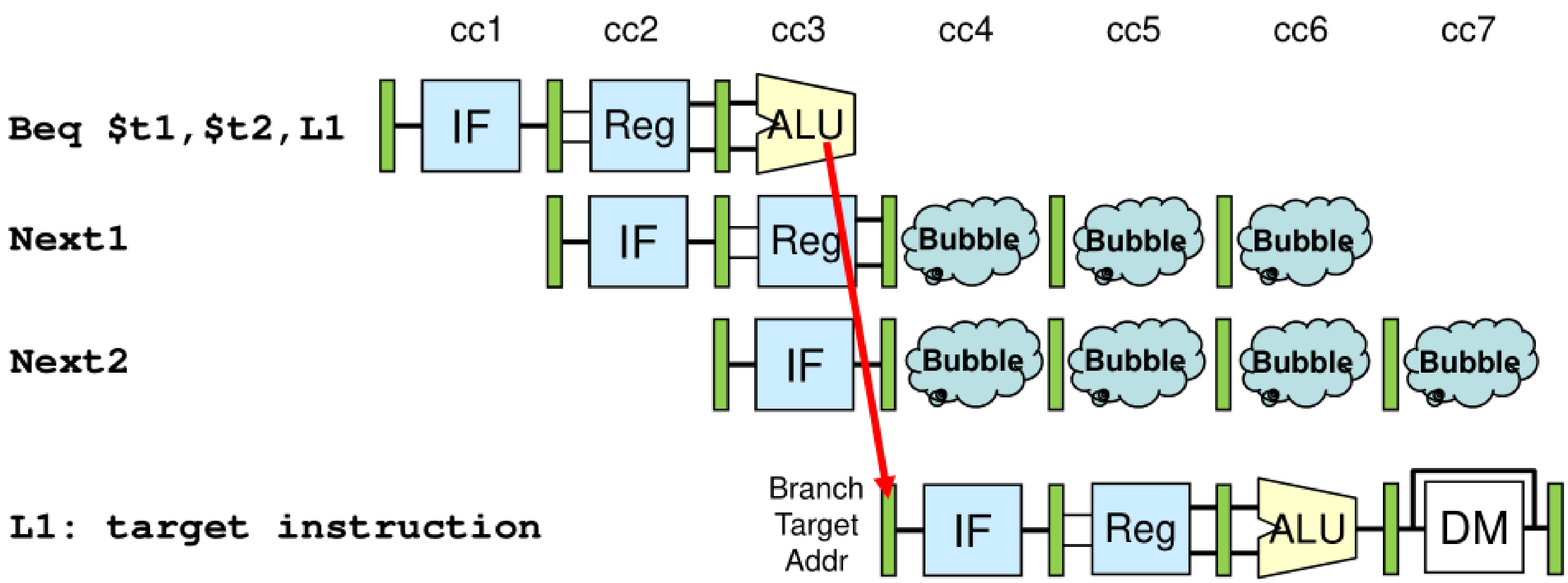
Example: pipeline after flushing



Another way to fix branch hazards

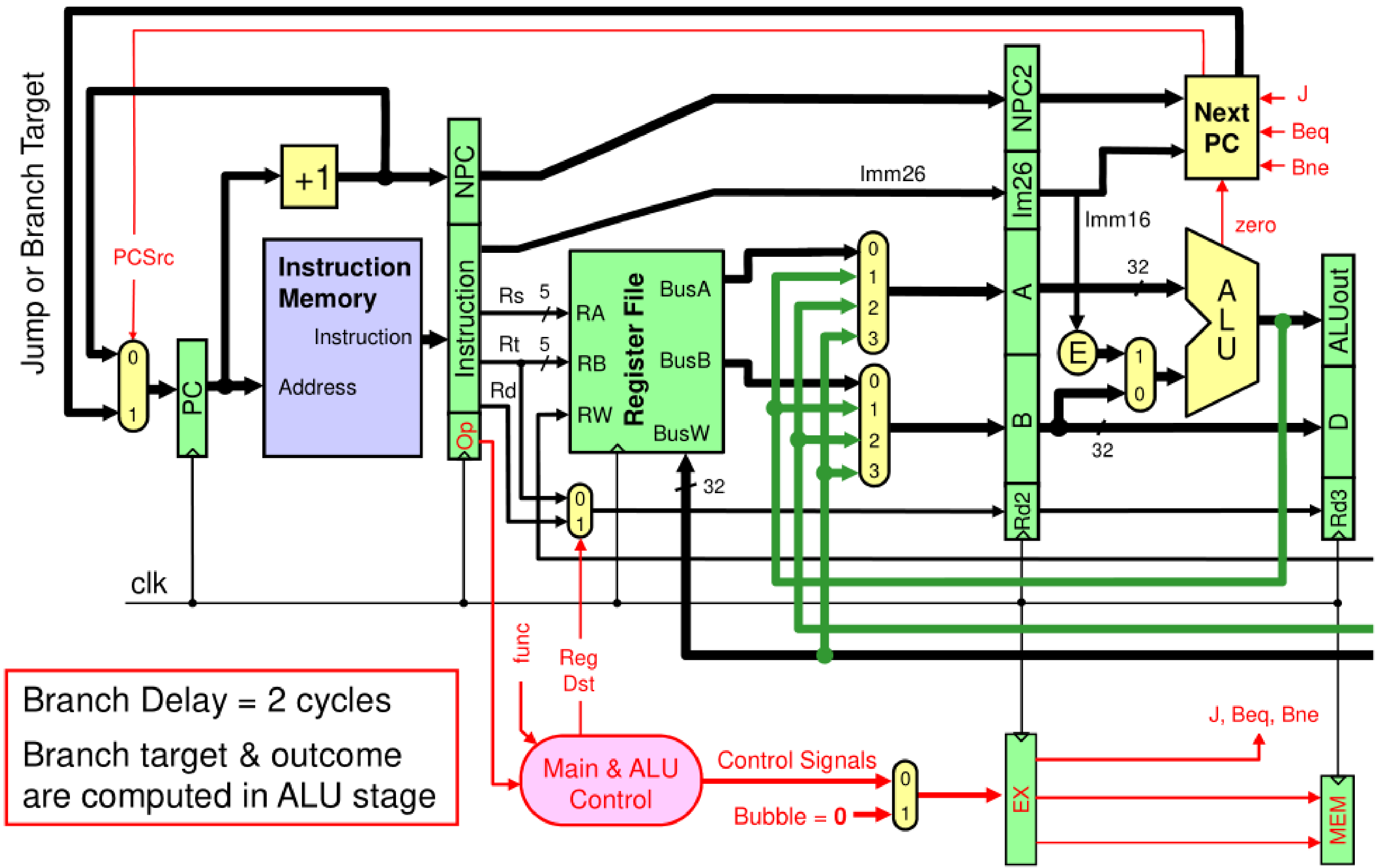
- Branch instruction needs two things:
  - **Branch Result**: taken or not taken
  - **Branch Target Address**: PC+4 (not taken) or PC+4+4×immediate (taken)
- Predict branches are **always not taken** – and take corrective action when wrong (i.e., taken)
  - Control logic detects a branch instruction in the 2^{nd} stage
  - ALU computes the **branch outcome** in the 3^{rd} stage
  - The **Next1** and **Next2** instructions will be fetched anyway.
  - If the branch is **taken** → **Next1** and **Next2** must be discarded (**flushed**) by converting them to bubbles → **wasted 2 cycles**.
  - If branches are untaken, proceed as normal → save the cost of control hazard.

2-Cycle Branch Delay Illustration



- Branch is **not taken**
  - The **Next1** and **Next2** instructions have been fetched → already in the pipeline → **flushed** (affects CPI).
  - To flush instructions in the IF stage, we add a control line, called IF.Flush, that zeros the instruction field of the IF/ID pipeline register.

Implementing branch prediction

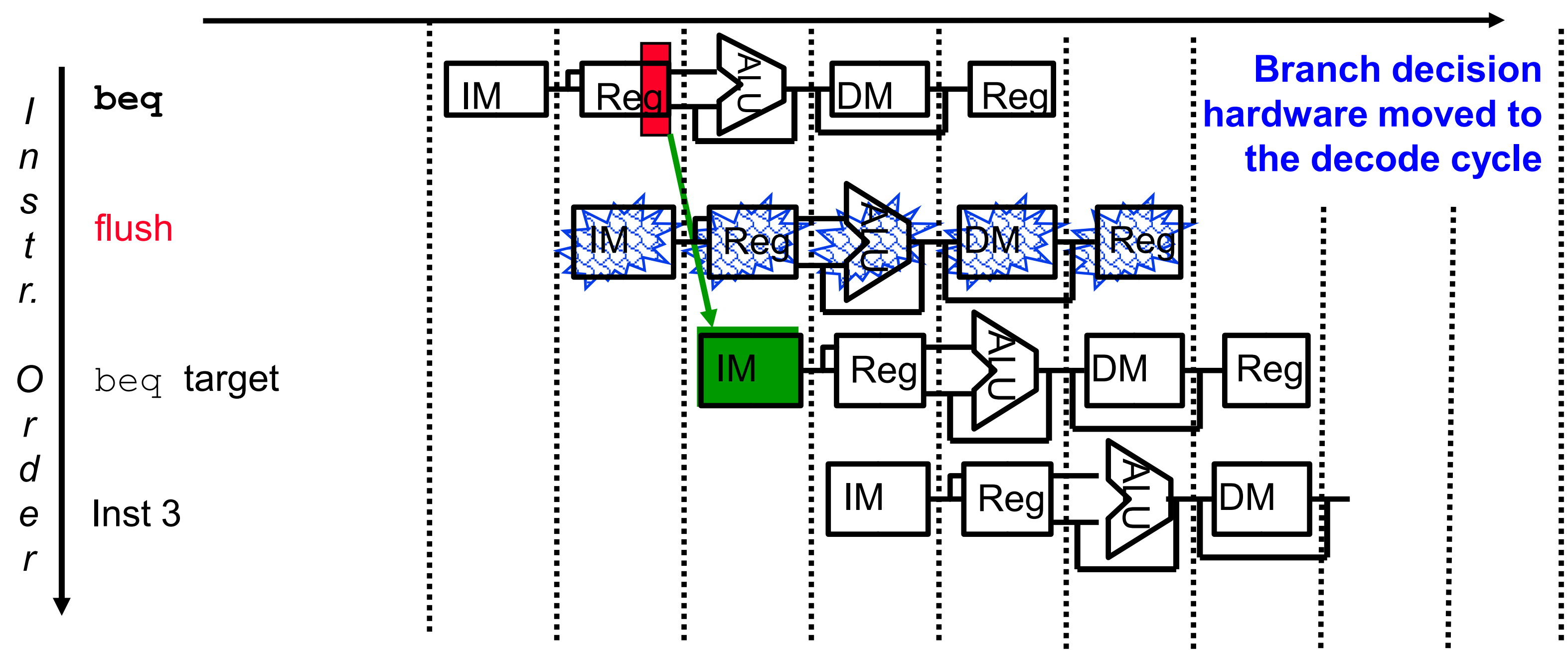


Branch Delay = 2 cycles

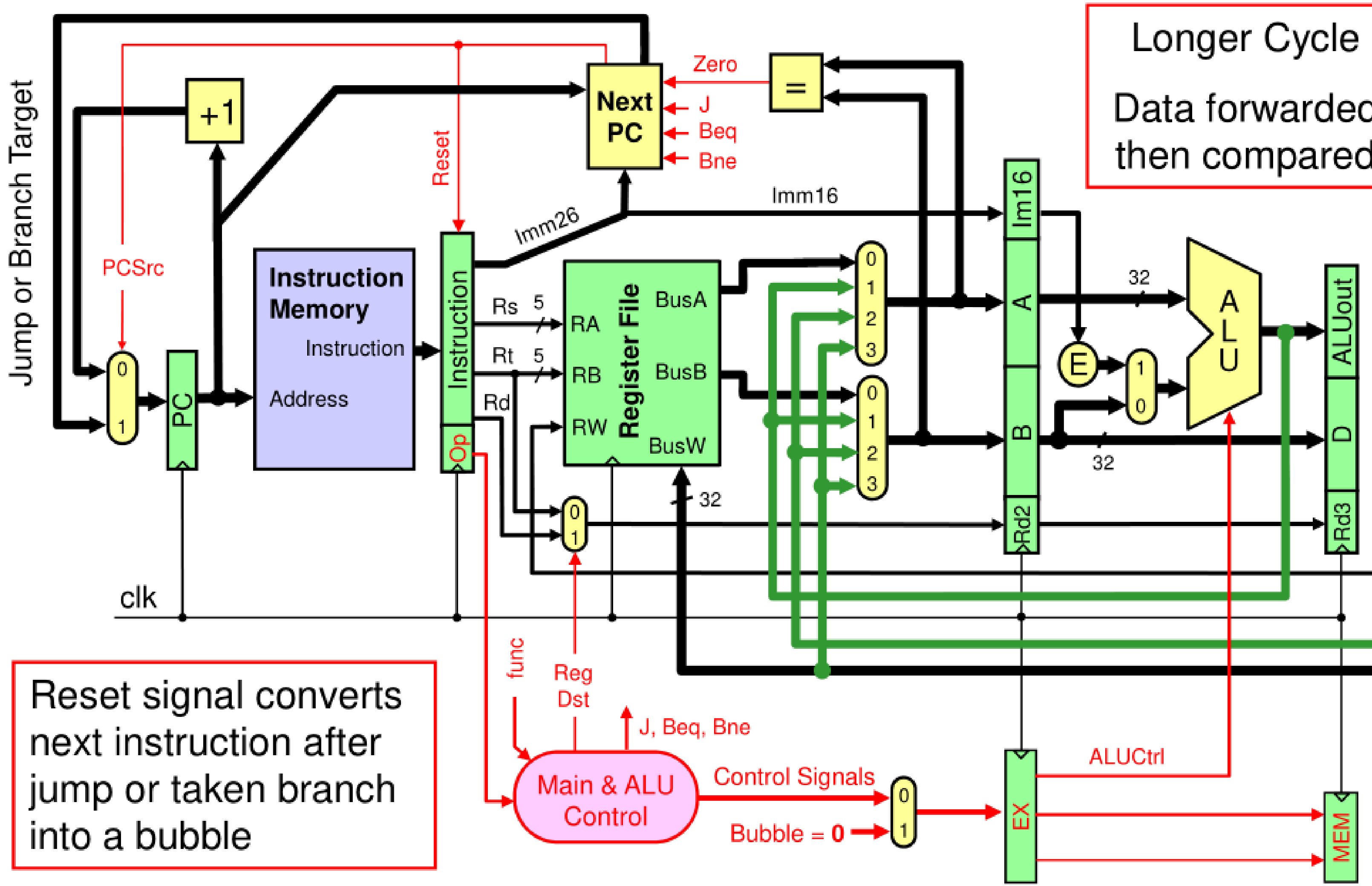
Branch target & outcome are computed in ALU stage

Reducing the Delay of Branches

- Move branch decision back to as **early** in the pipeline as possible – i.e., during the decode cycle
  - Need extra hardware to test registers, calculate the branch address, and update the PC during the second stage of the pipeline → one cycle delay.



Improved branch prediction implementation



Longer Cycle

Data forwarded then compared

Reset signal converts next instruction after jump or taken branch into a bubble

Further improvement: introducing delay slots

branch instruction	
branch delay slot	(next instruction)
branch target	(if branch taken)

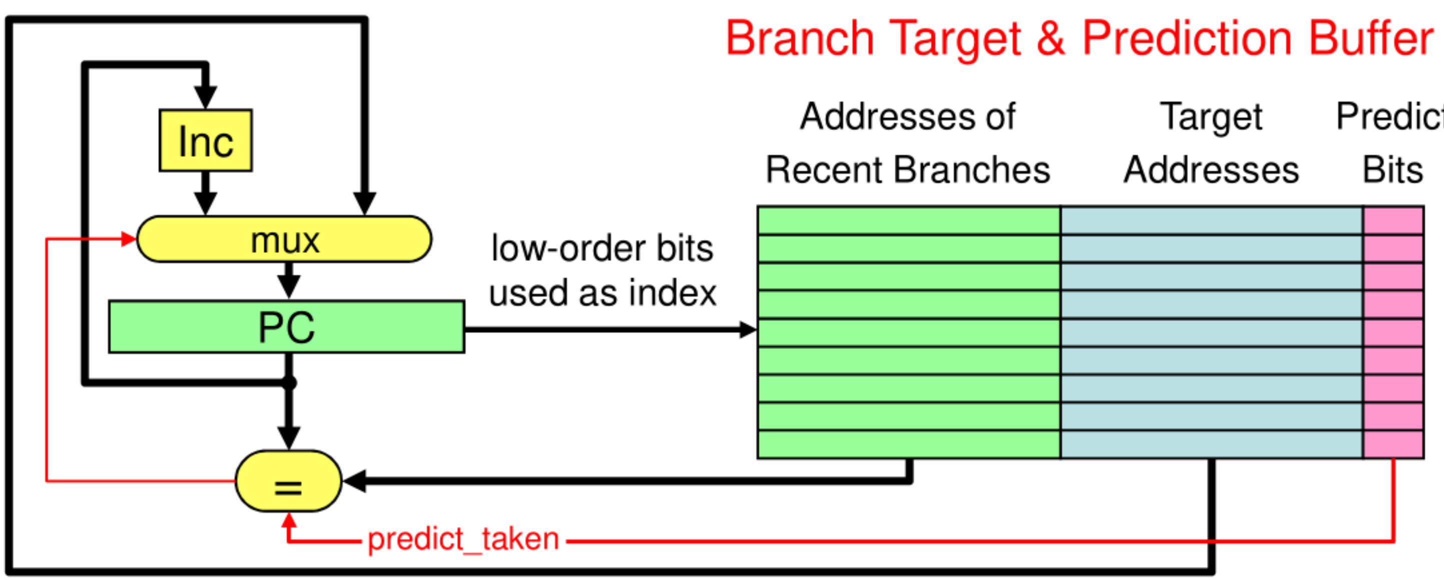
- Since we need to have a dead cycle anyway, let's put a useful instruction there → potentially get rid of all branch stalls.
  - For a 1-cycle branch delay, we have one **delay slot**.
  - MIPS compiler fills the delay slot by moving an instruction that is not affected by the branch to immediately after the branch thereby hiding the delay.
  - If **no independent** instruction is found, compiler fills delay slot with a **nop**.
- As processors go to both longer pipelines and issuing multiple instructions per clock cycle → the branch delay becomes longer
  - A single delay slot is insufficient → delayed branching has lost popularity compared to more expensive but more flexible **dynamic** approaches.

Dynamic Branch Prediction

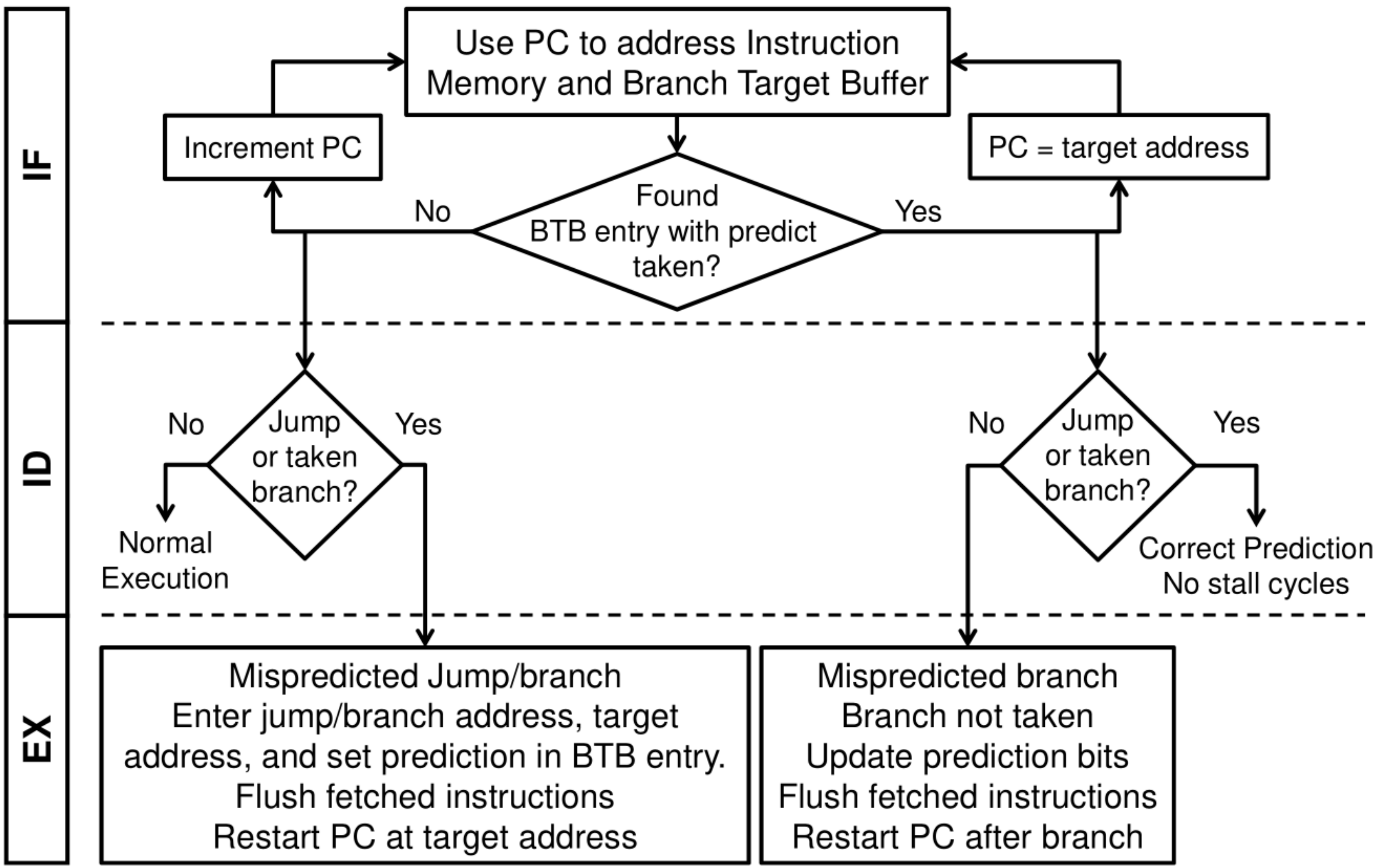
- In deeper and superscalar pipelines, branch penalty is more significant.
- Use dynamic prediction
  - **Branch target table** (BTB, aka BHT-branch history table)
    - Indexed by the lower portion of **recent** branch instruction addresses
    - Stores outcomes (taken/not taken)
  - To execute a branch
    - Check table, **expect the same outcome** as a previous branch branch that has the same low-order address bits.
    - Start fetching from fall-through or target, if true → no wasted cycles = **zero-delayed** branching!
    - If wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and **flip prediction**.
- A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)
  - 4096 bit BHT requires a lot of hardware.

Branch Target Buffer

- A small cache that stores the target addresses of recent branches and jumps.
- Also have prediction bits to predict whether branches are taken or not taken
  - The prediction bits are **dynamically determined by the hardware**.

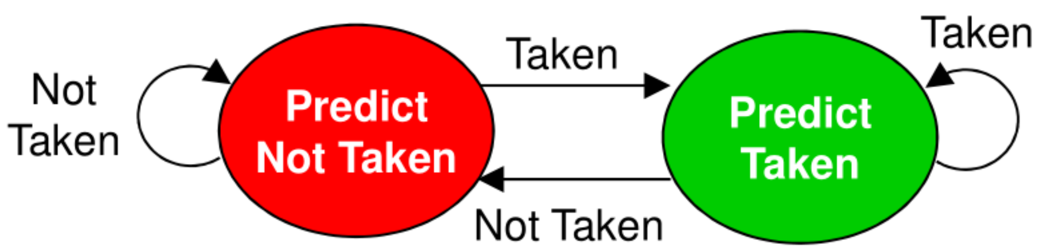


Dynamic Branch Prediction Flowchart



1-bit Prediction Scheme

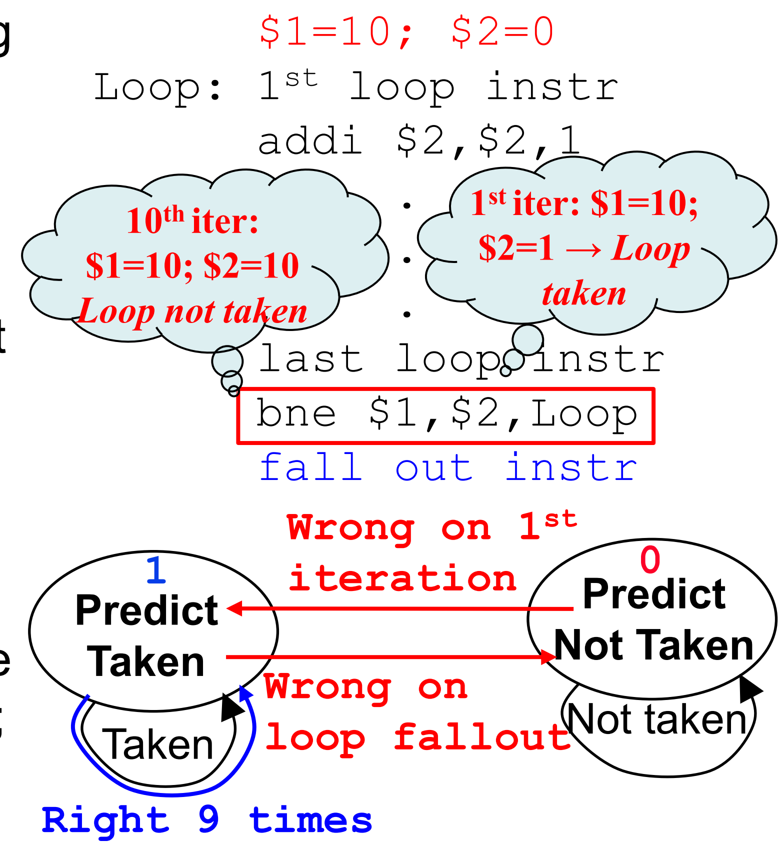
- Prediction is just a hint that is assumed to be correct.
- If incorrect then fetched instructions are flushed
- 1-bit prediction scheme is simplest to implement
  - 1 bit per branch instruction (associated with BTB entry)
  - Record last outcome of a branch instruction (Taken/Not taken)
  - Use last outcome to predict future behavior of a branch



1-Bit Predictor: Shortcoming

- A 1-bit predictor will be incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code

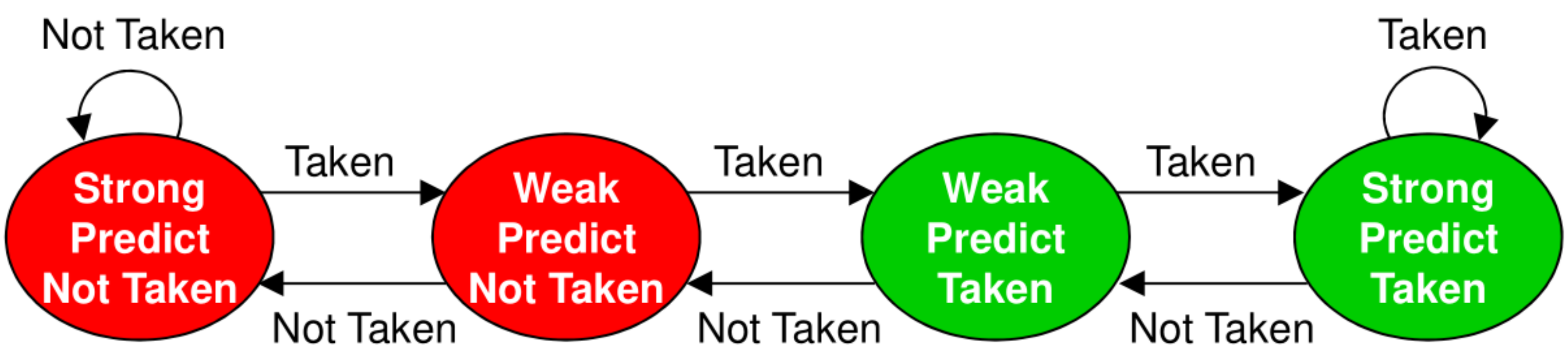
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
  2. As long as branch is taken (looping), prediction is correct
  3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)



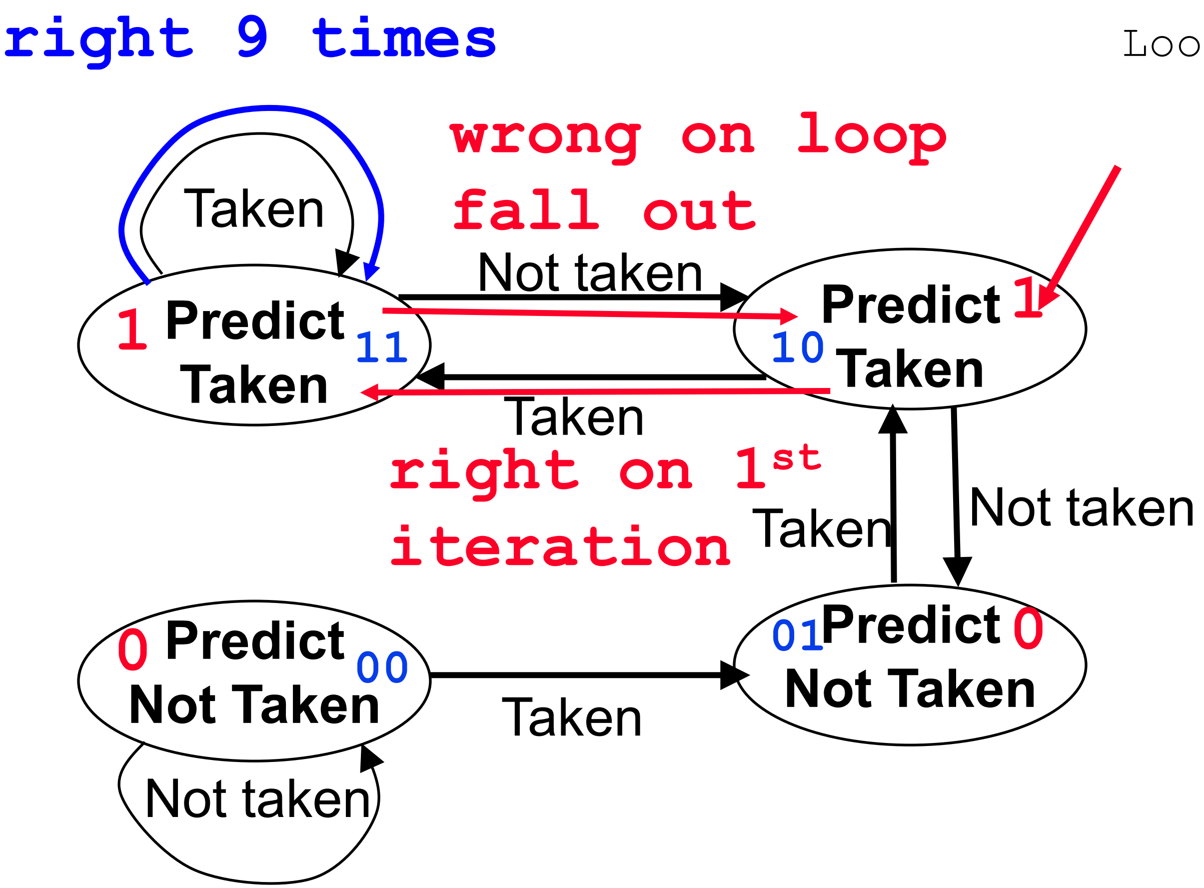
- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

2-bit Prediction Scheme

- 1-bit prediction scheme has a performance shortcoming.
- 2-bit prediction scheme works better and is often used
  - 4 states: strong and weak predict taken/predict not taken
- Implemented as a **saturating counter**
  - Counter is incremented to max=3 when branch outcome is taken
  - Counter is decremented to min=0 when branch is not taken



2-bit saturating counter predictor for the previous example




```
Loop: 1st loop instr
      2nd loop instr
          .
          .
          .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed