Pipeline Hazards

Computer Architecture

Five Stages of Pipeline

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.

The stages of instruction execution / pipelining are

- IF --- Instruction Fetch
- ID --- Instruction Decode / Register Read
- EX --- Execute in ALU / calculate address
- MEM --- Data memory access
- WB ---- Write back in register

The data flows from the left stage to right stage.

But in WB the result is written back (right to left data flow) in the register

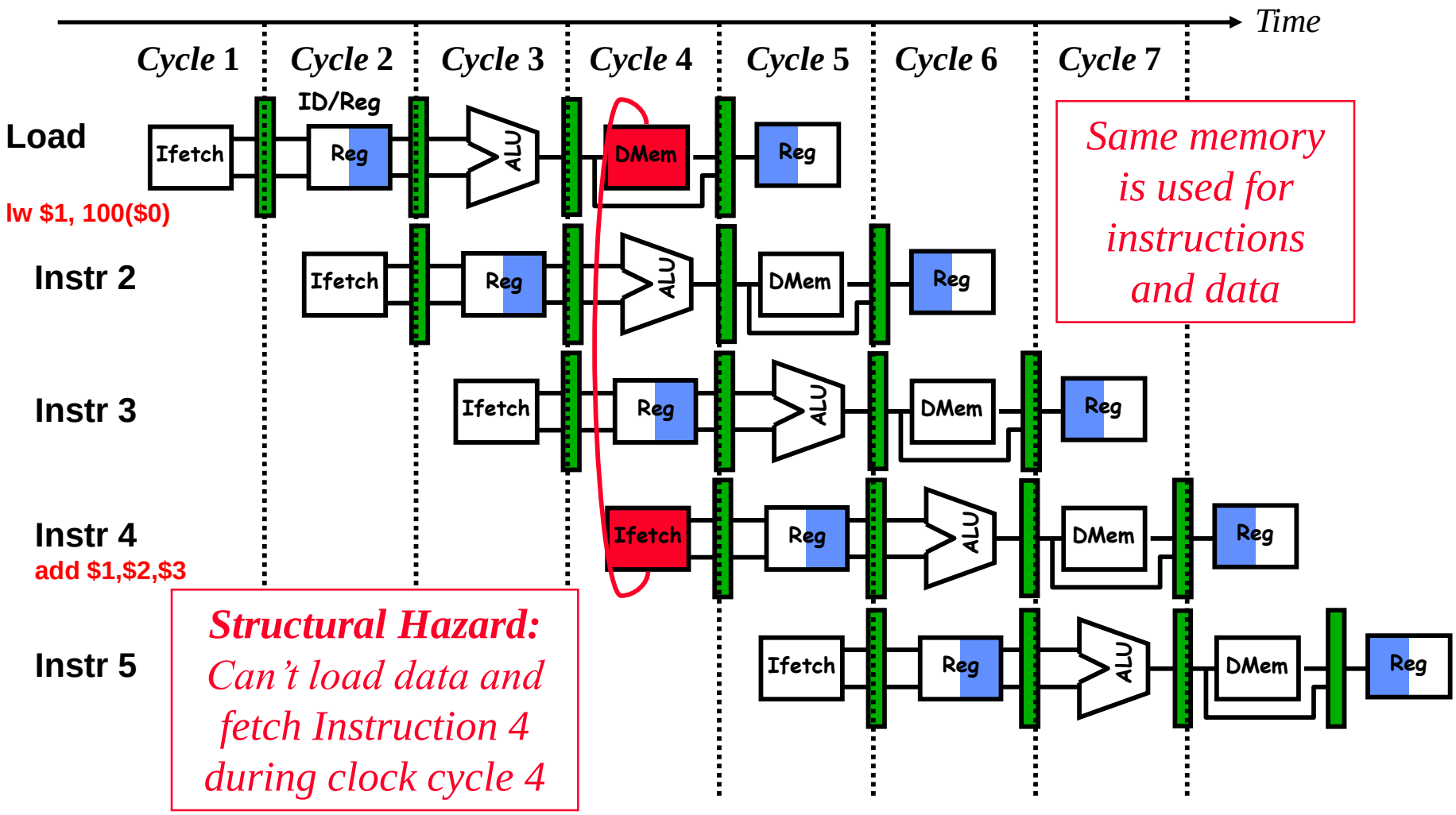
Pipeline Hazards

- Hazards: situations that makes the pipeline to stall or idle.

1. Structural hazards
   - Caused by resource contention
   - Using same resource by two instructions during the same cycle
2. Data hazards
   - An instruction may compute a result needed by next instruction
   - Hardware can detect dependencies between instructions
3. Control hazards
   - Caused by instructions that change control flow (branches/jumps)
   - Delays in changing the flow of control

- Hazards complicate pipeline control and limit performance

1. Structural Hazard - Conflict due to Memory Access

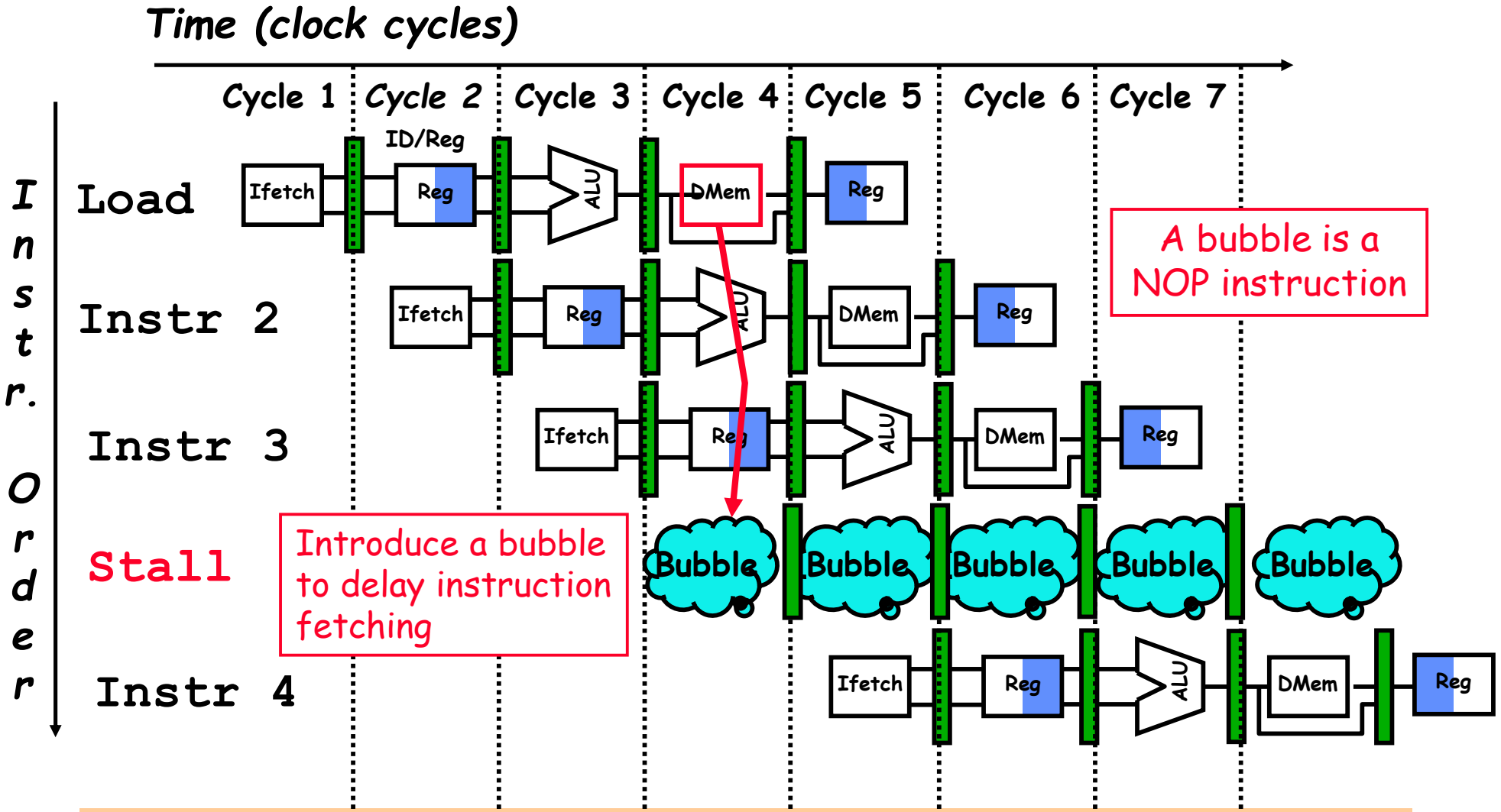


Resolving structural hazards

- Problem
  - Attempt to use the same hardware resource (Memory) by two different instructions during the same cycle
- Solution 1: Wait
  - Must detect the hazard
  - Must have mechanism to delay (stall) instruction access to resource (Introduce bubble / NOP)
  - Serious: hazard cannot be ignored
- Solution 2: Redesign the pipeline
  - Add more hardware to eliminate the structural hazard
  - In our example: use two memories with two memory ports
    - Instruction Memory
    - Data Memory

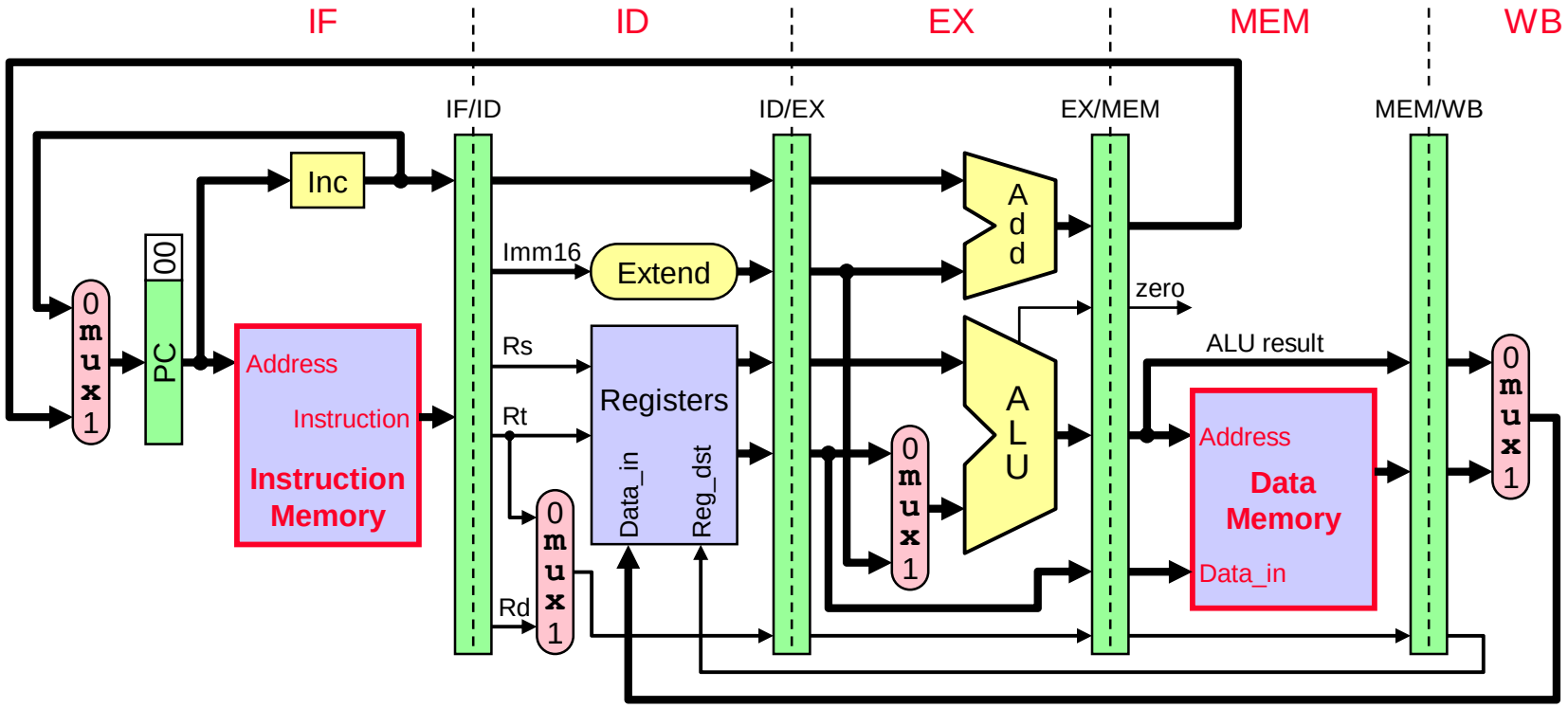
} Can be implemented as caches

Solution 1 : Detect Structural Hazard and Delay



Solution 2: Add More Hardware (Use Instruction and data memory)

- Eliminate structural hazard at design time
- Use two separate memories with two memory ports
  - Instruction and data memories can be implemented as caches



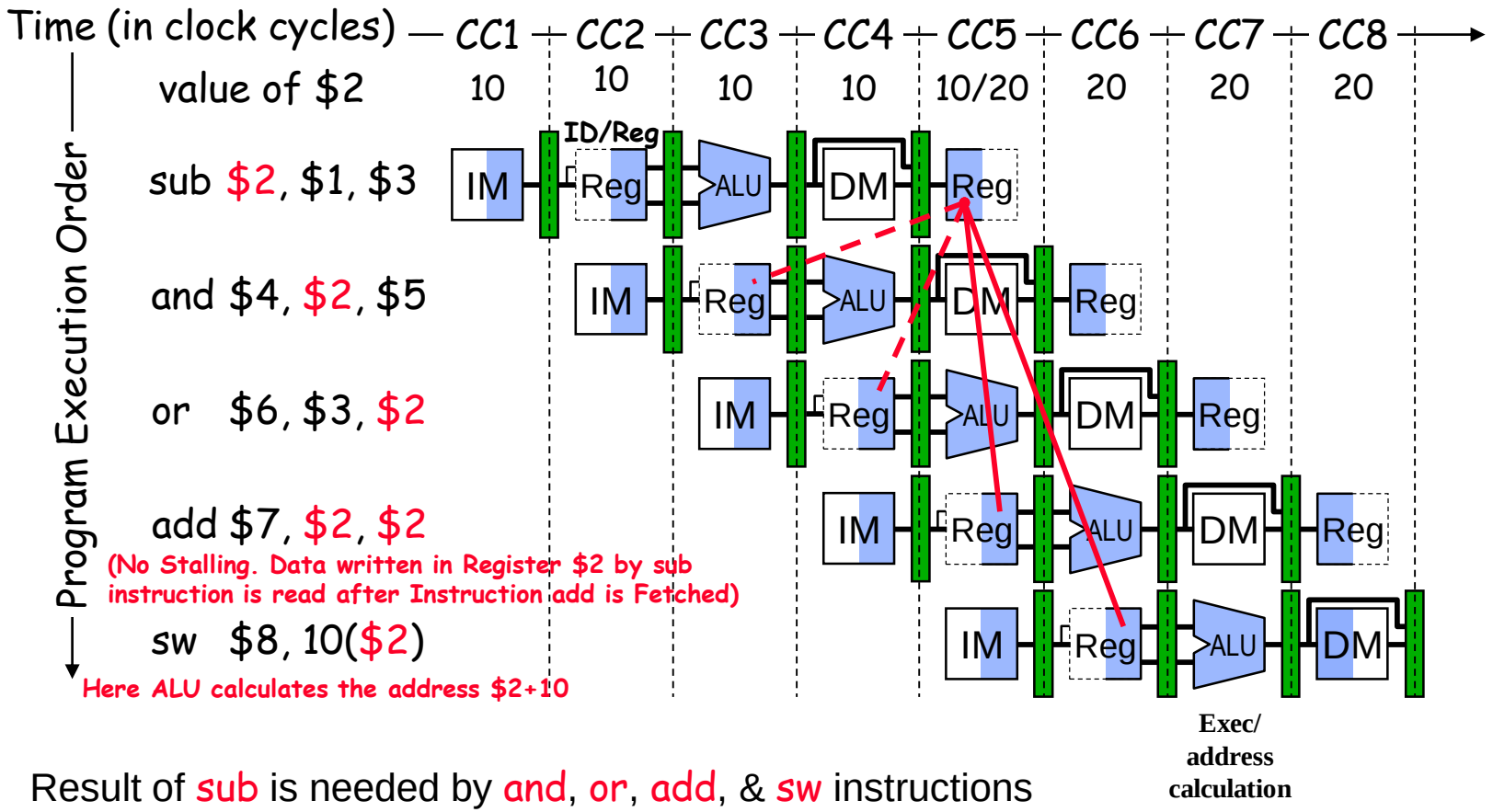
2. Data Hazards

- Dependency between instructions causes a data hazard
- The dependent instructions are close to each other
  - Pipelined execution might change the order of operand access
- Read After Write – RAW Hazard
  - Given two instructions *I* and *J*, where *I* comes before *J* …
  - Instruction *J* should read an operand after it is written by *I*
  - Called a data dependence in compiler terminology

```
I: add $1, $2, $3   # r1 is written
J: sub $4, $1, $3   # r1 is read
```

  - Hazard occurs when *J* reads the operand before *I* writes it

Example of a RAW Data Hazard



- Result of sub is needed by and, or, add, & sw instructions
- Instructions and & or will read old value of $2 from reg file
- During CC5, $2 is written and read – new value is read

Left highlight – write operation

Right highlight – Read operation

Blank – not used

2. Solutions to data hazard

a) Reordering code (Software)

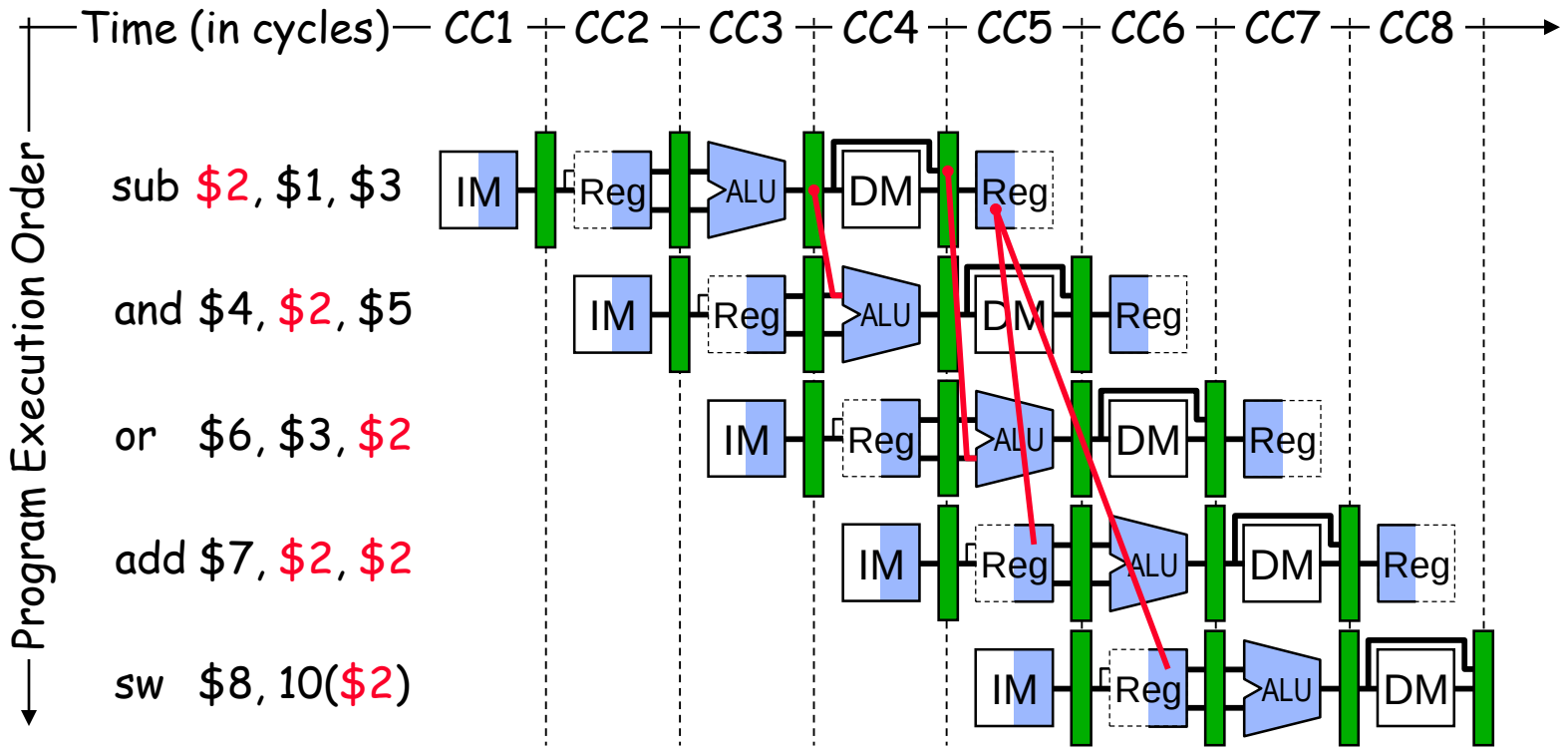
b) Operand forwarding (Hardware)

c) By using Stall

2a) Solution by reordering: The order of the instructions may be changed such that one instruction need wait for the other instruction's result without affecting the logic

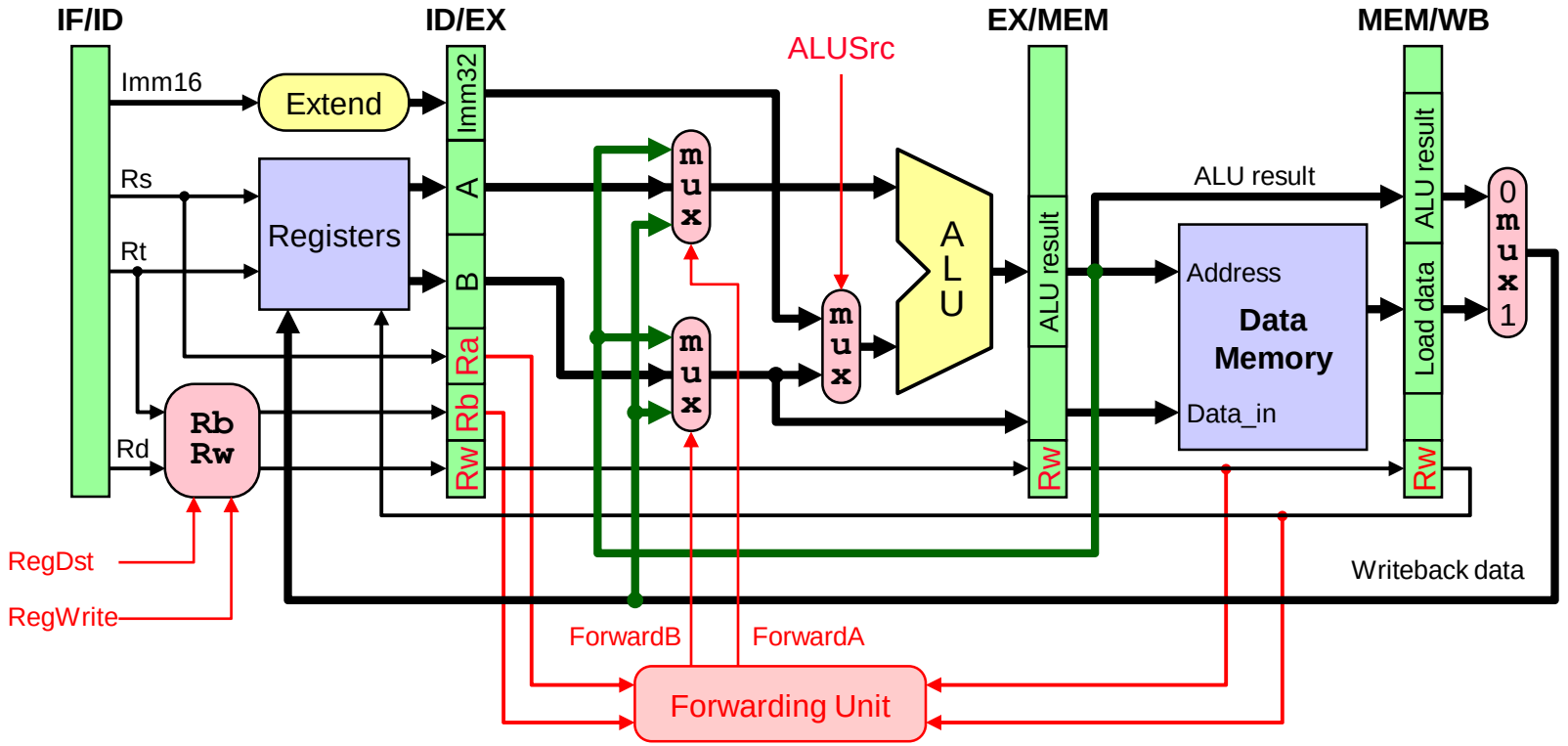
2b. Operand Forwarding (Forwarding ALU Result)

- The ALU result is forwarded (fed back) to the ALU input
  - No bubbles are inserted into the pipeline and no cycles are wasted
- ALU result exists in either EX/MEM or MEM/WB register

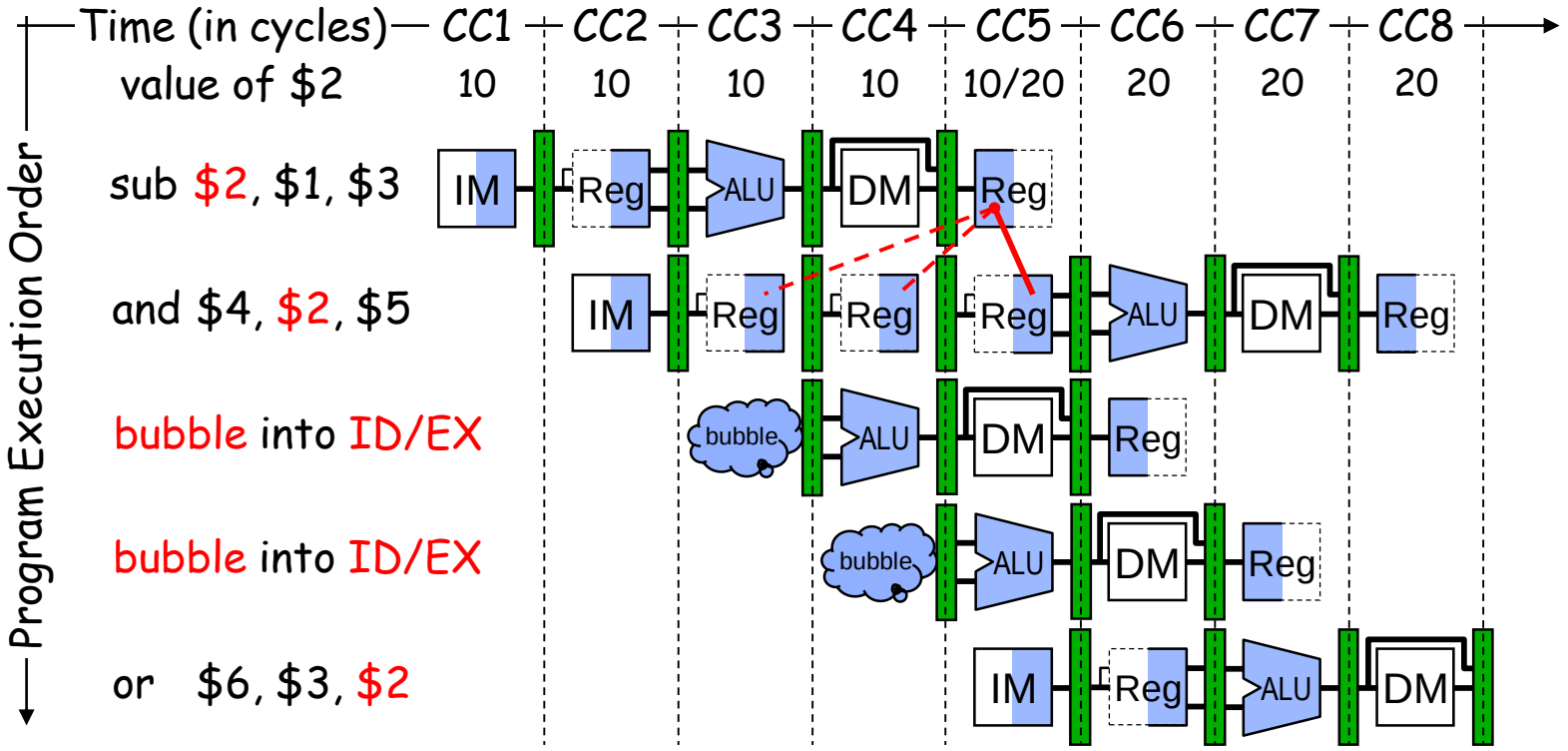


2b. Operand Forwarding Unit

- Forwarding unit generates ForwardA and ForwardB
  - That are used to control the two forwarding multiplexers
- Uses Ra and Rb in ID/EX and Rw in EX/MEM & MEM/WB



2c. Stalling the Pipeline



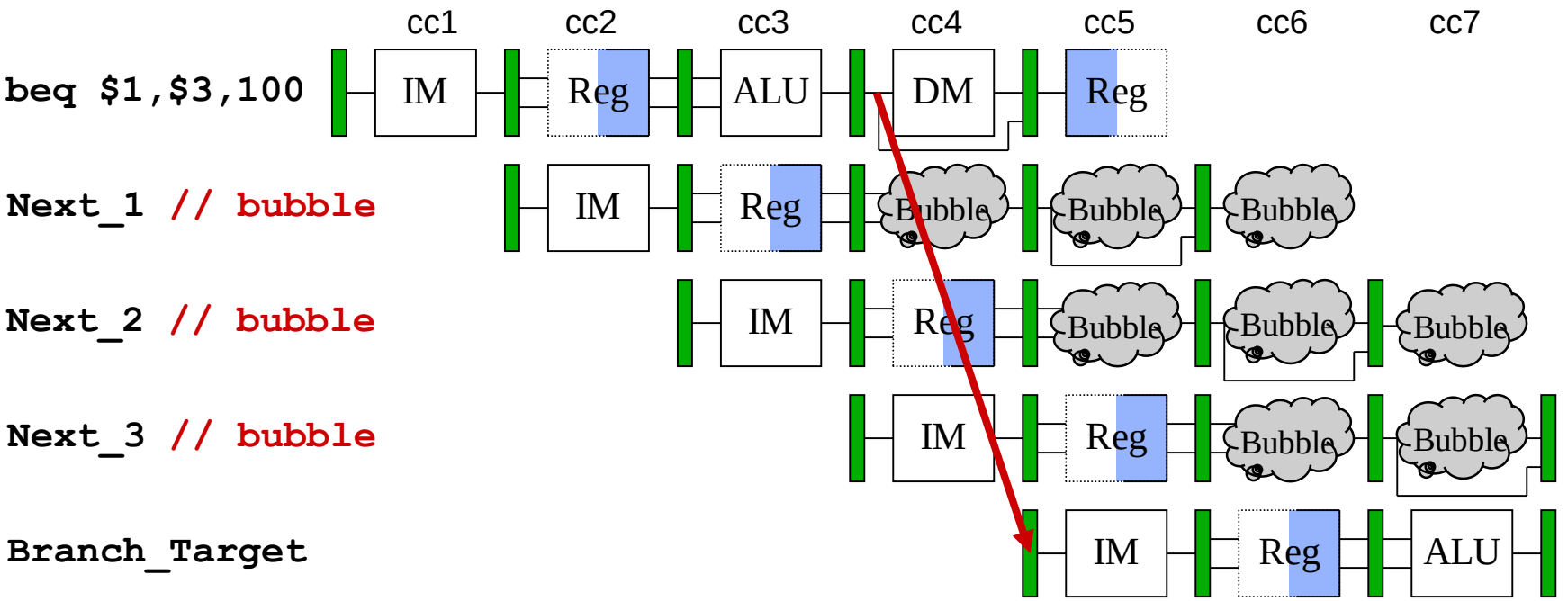
- The and instruction cannot fetch $2 until CC5
- Two bubbles (NOP instructions) are inserted in ID/EX
  at the end of CC3 and CC4 cycles

3. Control Hazards

- Branch instructions can cause great performance loss
- Branch instructions need two things:
  - The result of branch: Taken or Not Taken
  - Branch target address:
    - PC + 4 Branch NOT taken
    - PC + 4 + immediate*4 Branch Taken
- Branch instruction is not detected until the ID stage
  - At which point a new instruction has already been fetched
- For our original pipeline:
  - Effective address is not calculated until EX stage
  - Branch condition get set in the EX/MEM register (EX/MEM.zero)
  - 3-cycle branch delay

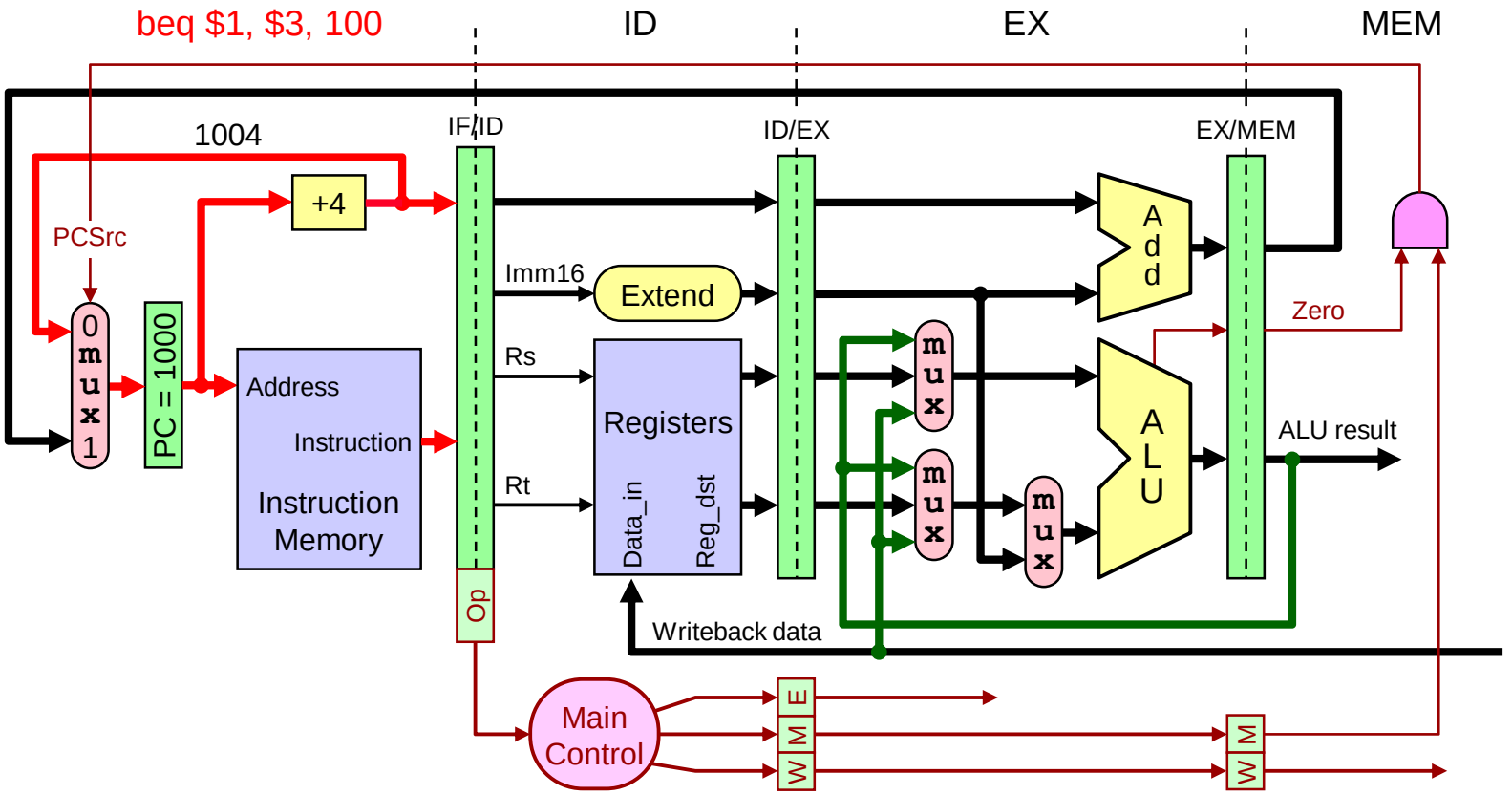
3-Cycle Branch Delay

- Instructions Next_1 thru Next_3 stored continuously with beq will be fetched anyway
- Pipeline should flush Next_1 thru Next_3 if branch is taken
- Otherwise, they can be executed normally



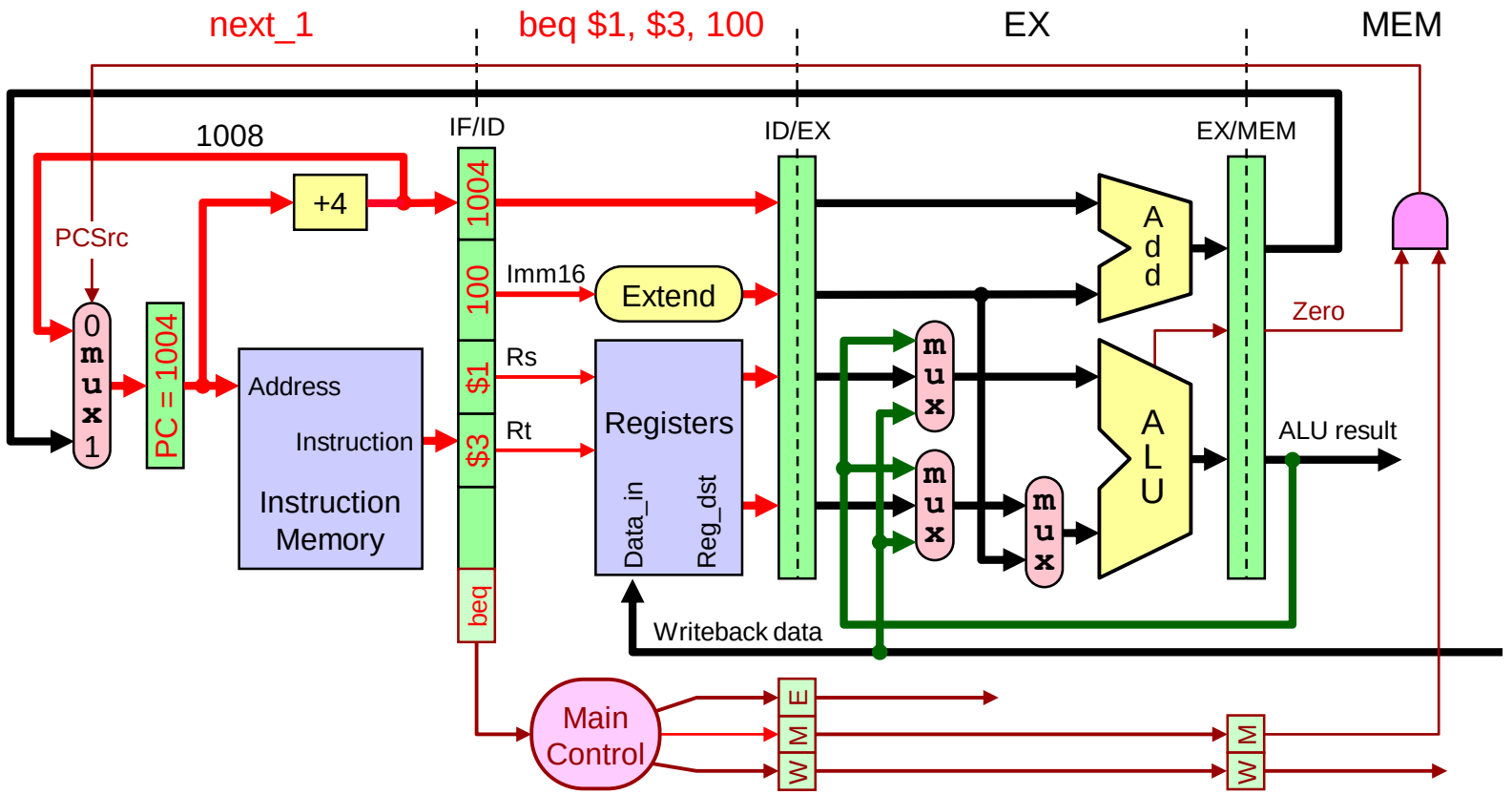
Branch Delay – CC1

- Consider the pipelined execution of: beq $1, $3, 100
- During the first cycle, beq is fetched in the IF stage



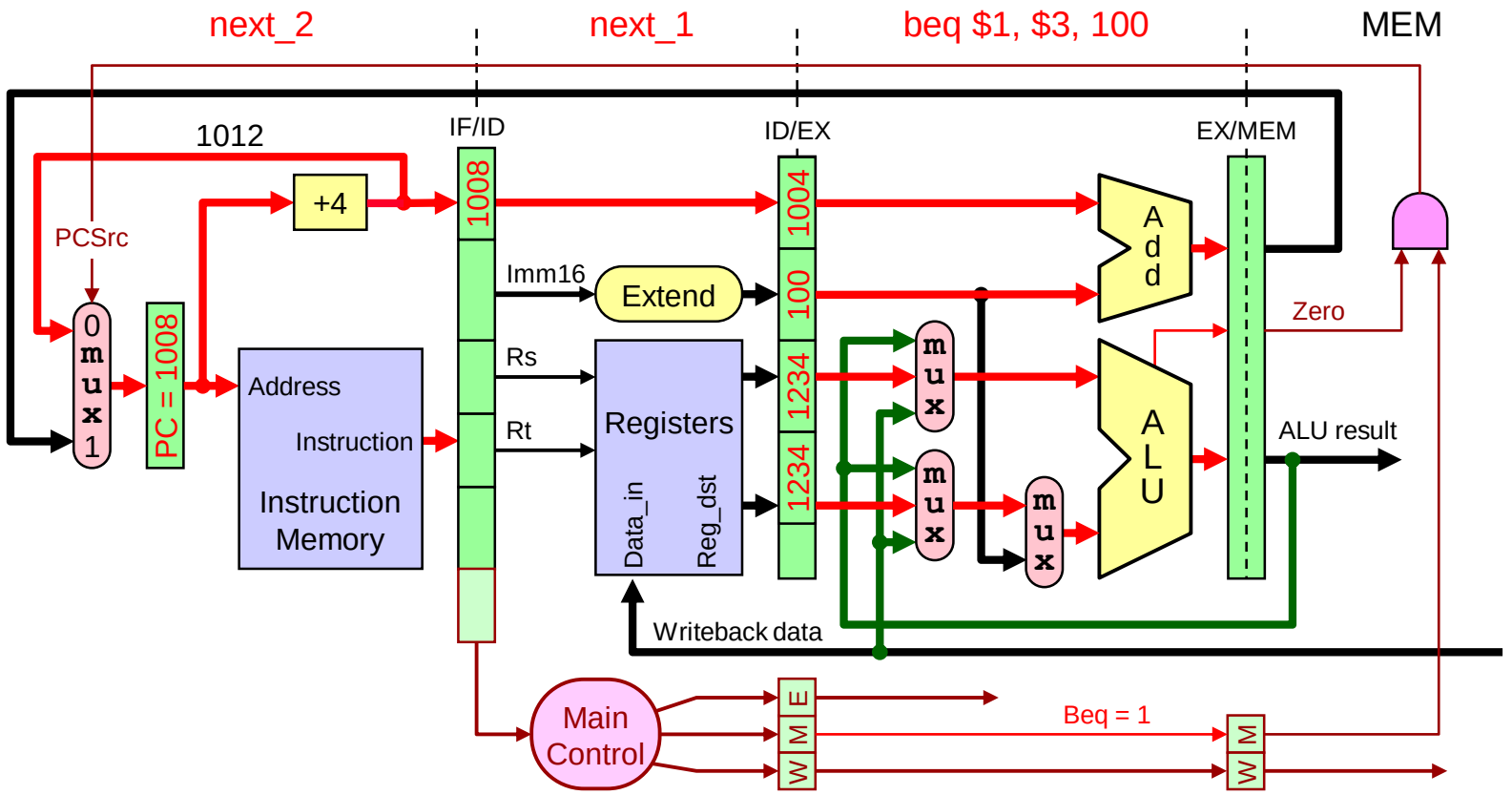
Branch Delay – CC2

- During the second cycle, beq is decoded in the ID stage
- The next_1 instruction is fetched in the IF stage



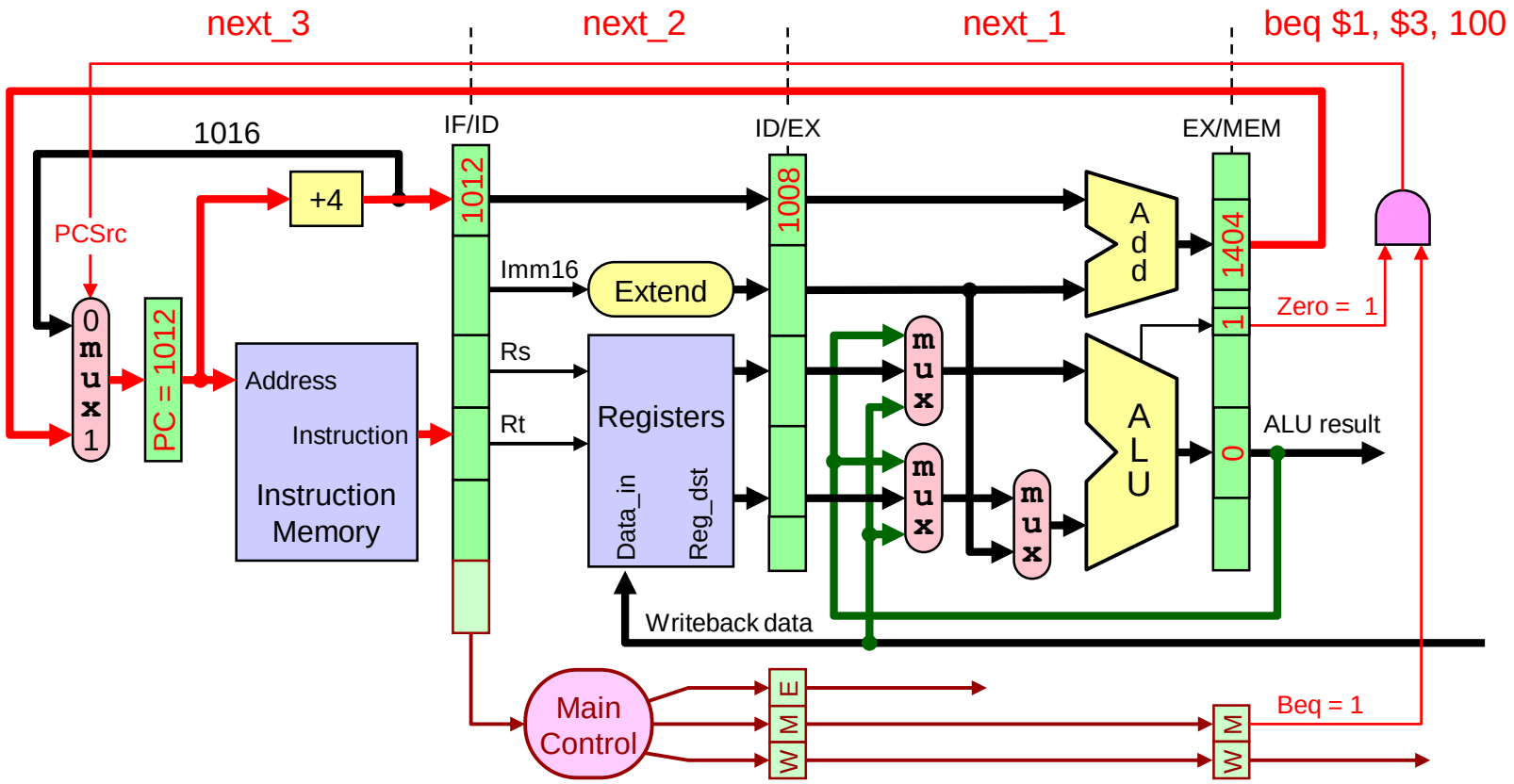
Branch Delay – CC3

- During the third cycle, beq is executed in the EX stage
- The next_2 instruction is fetched in the IF stage



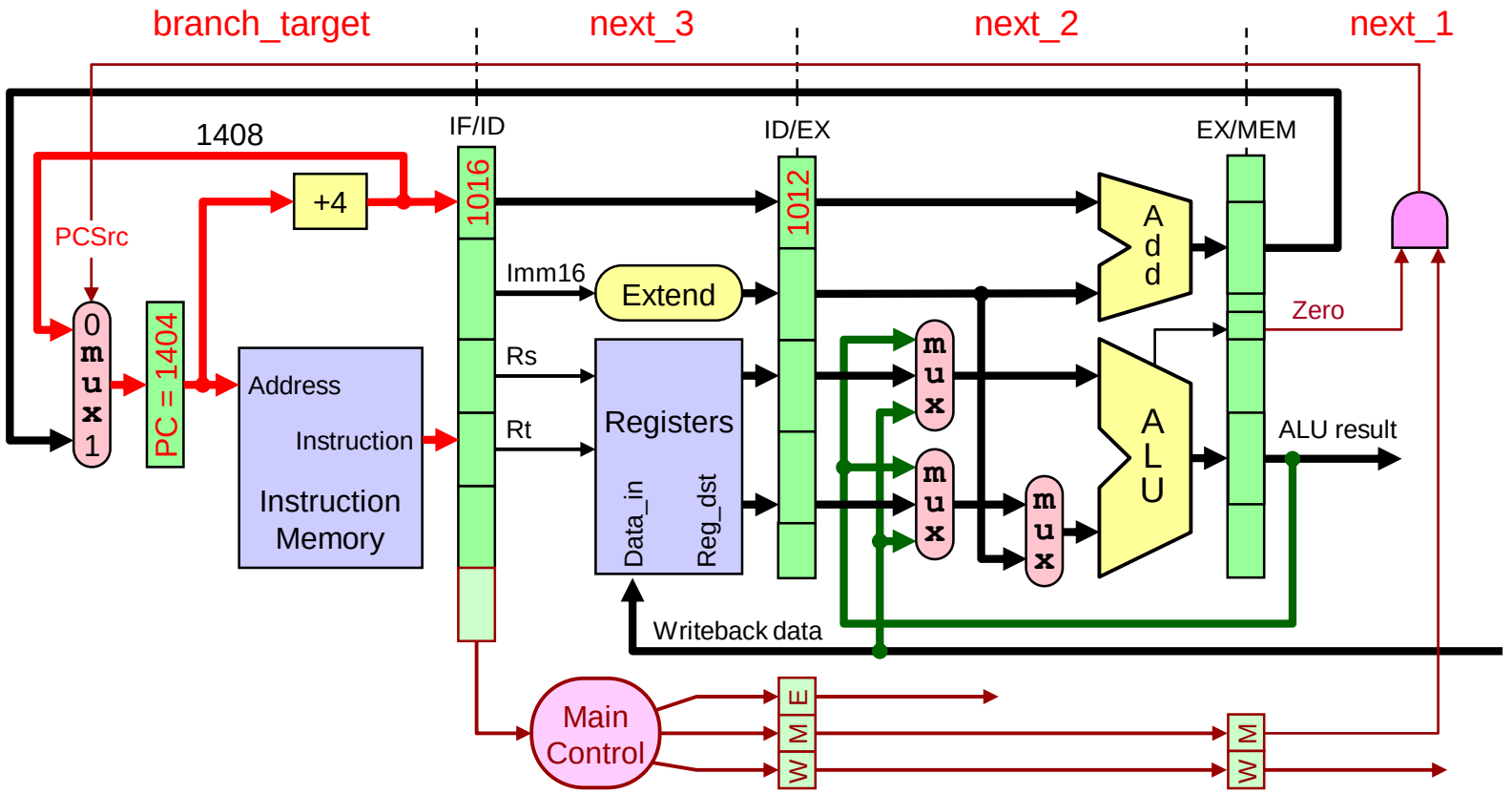
Branch Delay – CC4

- During the fourth cycle, beq reaches MEM stage
- The next_3 instruction is fetched in the IF stage



Branch Delay – CC5

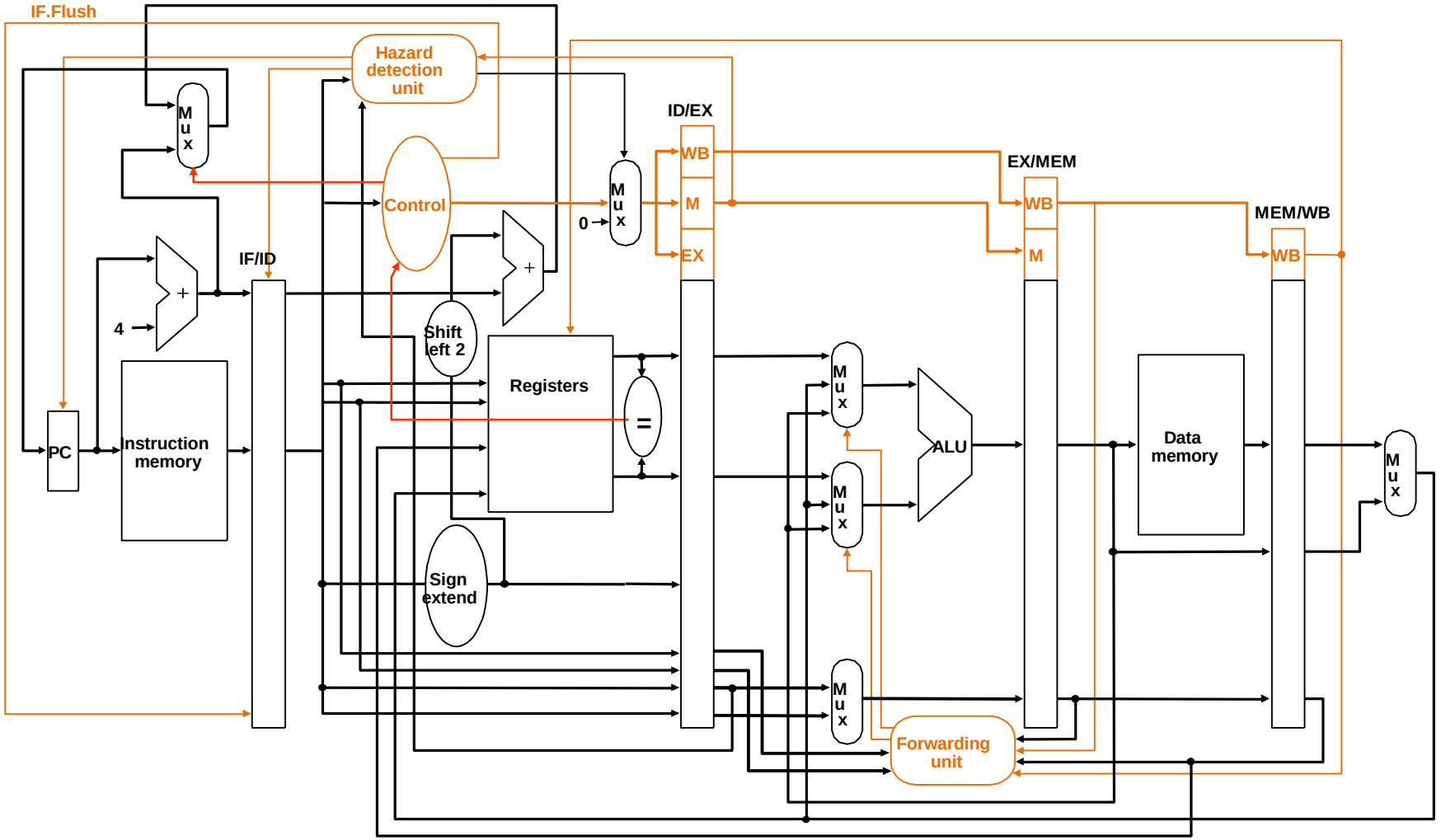
- During the fifth cycle, branch_target instruction is fetched
- Next_1 thru next_3 should be converted into NOPs



Reducing the Delay of Branches

- Branch delay can be reduced from 3 cycles to just 1 cycle
- Branch decision is moved from 4th into 2nd pipeline stage
  - Branches can be determined earlier in the ID stage
  - Branch address calculation adder is moved to ID stage
  - A comparator in the ID stage to compare the two fetched registers
    - To determine branch decision, whether the branch is taken or not
- Only one instruction that follows the branch will be fetched
- If the branch is taken then only one instruction is flushed
- We need a control signal IF.Flush to zero the IF/ID register
  - This will convert the fetched instruction into a NOP

Reducing the Delay of Branches



Branch Hazard Alternatives

- Always stall the pipeline until branch direction is known
  - Next instruction is always flushed (turned into a NOP)
- Predict Branch Not Taken
  - Fetch successor instruction: PC+4 already calculated
  - Almost half of MIPS branches are not taken on average
  - Flush instructions in pipeline only if branch is actually taken
- Predict Branch Taken
  - Can predict backward branches in loops $\Rightarrow$ taken most of time
  - However, branch target address is determined in ID stage
  - Must reduce branch delay from 1 cycle to 0, but how?
- Delayed Branch
  - Define branch to take place AFTER the following instruction

Delayed Branch

- Define branch to take place after the next instruction
- For a 1-cycle branch delay, we have one delay slot

  branch instruction

  branch delay slot – next instruction

  . . .

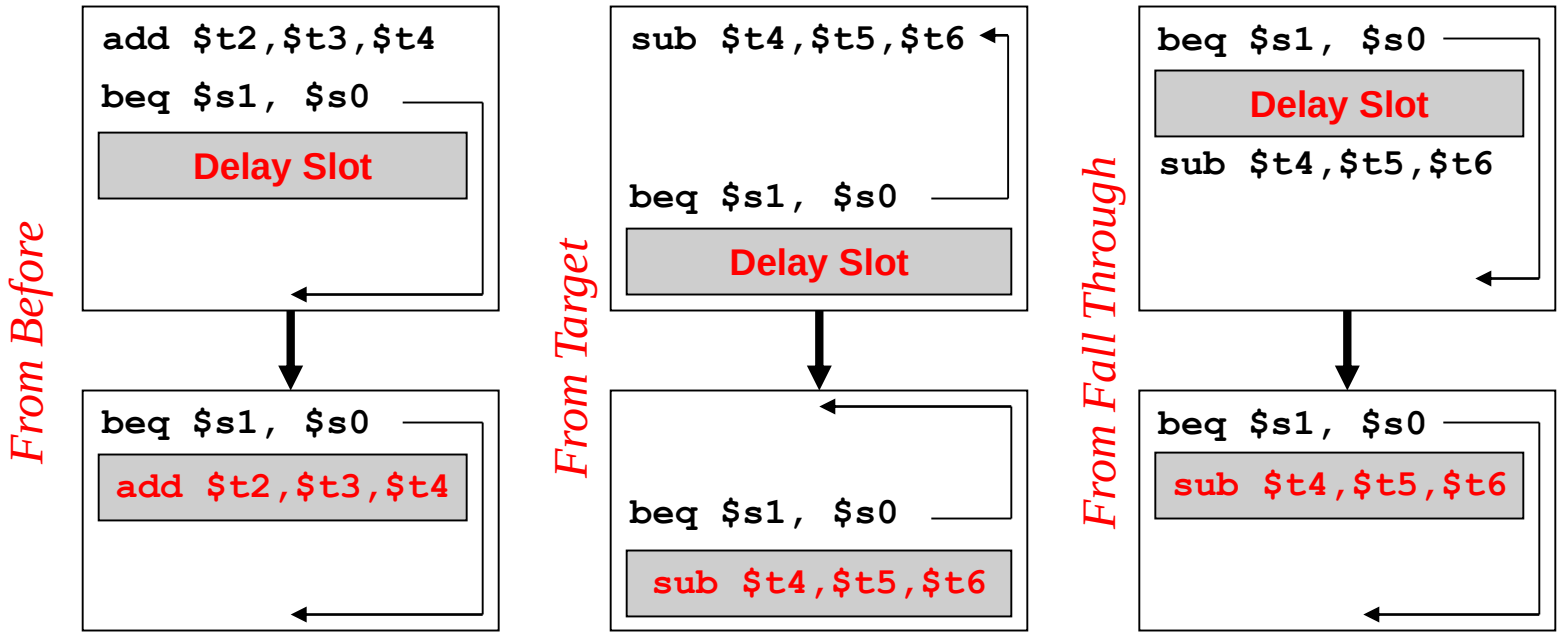
  branch target – if branch taken

branch instruction (taken)	**IF**	**ID**	**EX**	**MEM**	**WB**	
branch delay slot (next instruction)		**IF**	**ID**	**EX**	**MEM**	**WB**
branch target			**IF**	**ID**	**EX**	**MEM**	**WB**

- Compiler/assembler fills the branch delay slot
  - By selecting a useful instruction

Scheduling the Branch Delay Slot

- From an independent instruction before the branch
- From a target instruction when branch is predicted taken
- From fall through when branch is predicted not taken



More on Delayed Branch

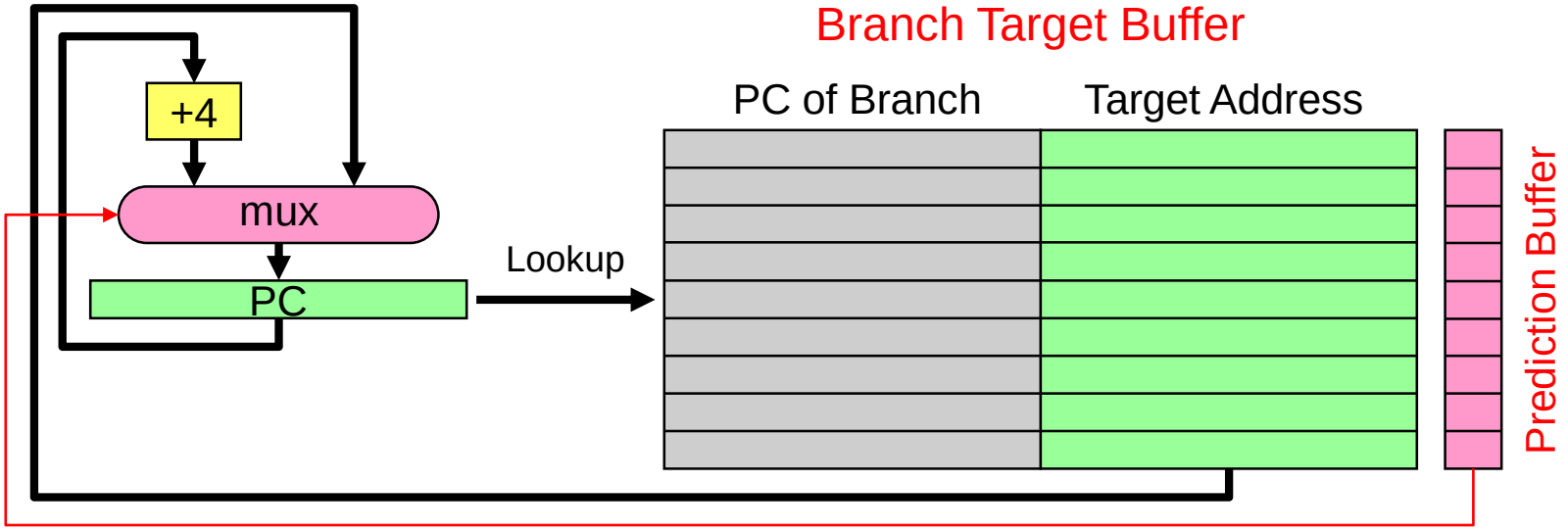
- Scheduling delay slot with
  - Independent instruction is the best choice
    - However, not always possible to find an independent instruction
  - Target instruction is useful when branch is predicted taken
    - Such as in a loop branch
    - May need to duplicate instruction if it can be reached by another path
    - Cancel branch delay instruction if branch is not taken
  - Fall through is useful when branch is predicted not taken
    - Cancel branch delay instruction if branch is taken
- Disadvantages of delayed branch
  - Branch delay can increase to multiple cycles in deeper pipelines
  - Zero-delay branching + dynamic branch prediction are required

Zero-Delayed Branch

- How can we achieve zero-delay for a taken branch …
  - If the branch target address is computed in the ID stage ?
- Solution
  - Check the PC to see if the instruction being fetched is a branch
  - Store the branch target address in a table in the IF stage
  - Such a table is called the branch target buffer
  - If branch is predicted taken then
    - Next PC = branch target fetched from target buffer
  - Otherwise, if branch is predicted not taken then
    - Next PC = PC + 4
  - Zero-delay is achieved because Next PC is determined in IF stage

Branch Target and Prediction Buffer

- The branch target buffer is implemented as a small cache
  - That stores the branch target address of taken branches
- We also have a branch prediction buffer
  - To store the prediction bits for branch instructions
  - The prediction bits are dynamically determined by the hardware

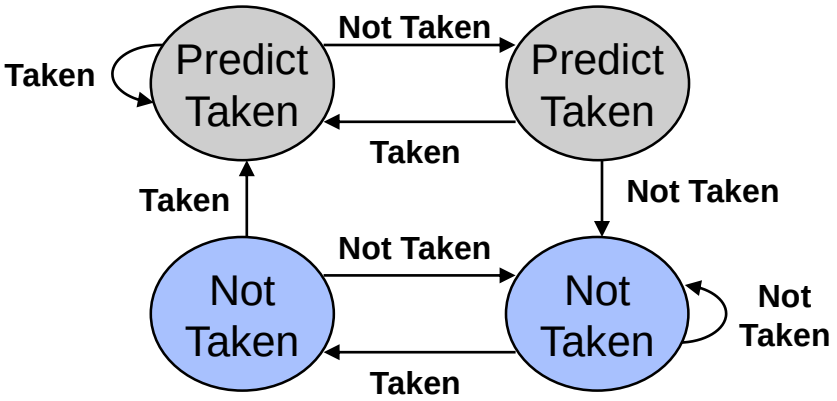


Dynamic Branch Prediction

- Prediction of branches at runtime using prediction bits
  - One or few prediction bits are associated with a branch instruction
- Branch prediction buffer is a small memory
  - Indexed by the lower portion of the address of branch instruction
- The simplest scheme is to have 1 prediction bit per branch
- We don't know if the prediction bit is correct or not
- If correct prediction …
  - Continue normal execution – no wasted cycles
- If incorrect prediction (misprediction) …
  - Flush the instructions that were incorrectly fetched – wasted cycles
  - Update prediction bit and target address for future use

2-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 has a performance shortcoming
  - A loop branch is almost always taken, except for last iteration
  - 1-bit scheme will predict incorrectly twice, rather than once
  - On the first and last loop iterations
- 2-bit prediction schemes are often used
  - A prediction must be wrong twice before it is changed
  - A loop branch is mispredicted only once on the last iteration



Implementing Forwarding

- Two multiplexers are added at the inputs of the ALU
- ALU result in EX/MEM is forwarded (fed back)
- Writeback data in MEM/WB is also forwarded
- Two signals: ForwardA and ForwardB control forwarding