control hazard in pipeline and different solution technique

Presented By

Dr. Banchhanidhi Dash

School of Computer Engineering
KIIT University

Control Hazards

A *control hazard* is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

Control hazards arise when the CPU cannot determine which instruction to fetch next.

We can minimize delays by doing branch tests earlier in the pipeline.

We can also take a chance and predict the branch direction, to make the most of a bad situation.

control Hazard Solutions technique

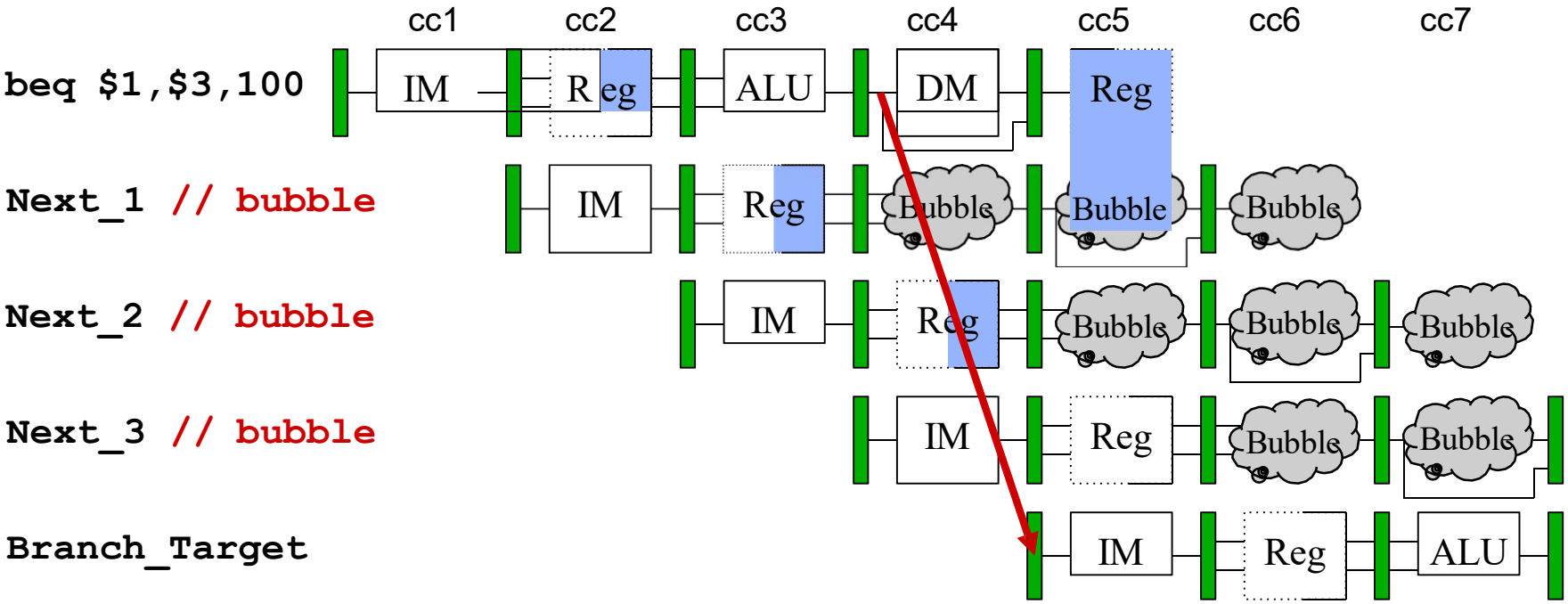
- **Stalling - stop fetching instr. until result is available**
  - **Significant performance penalty**
  - **Hardware required to stall**
- **Delayed branch - specify in architecture that following instruction is always executed**
  - **Compiler re-orders instructions into delay slot**
  - **Insert "NOP" (no-op) operations when can't use**
- **branch Prediction - assume an outcome and continue fetching (undo if prediction is wrong)**
  - **Performance penalty only when guess wrong**
  - lose cycle only when misprediction

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

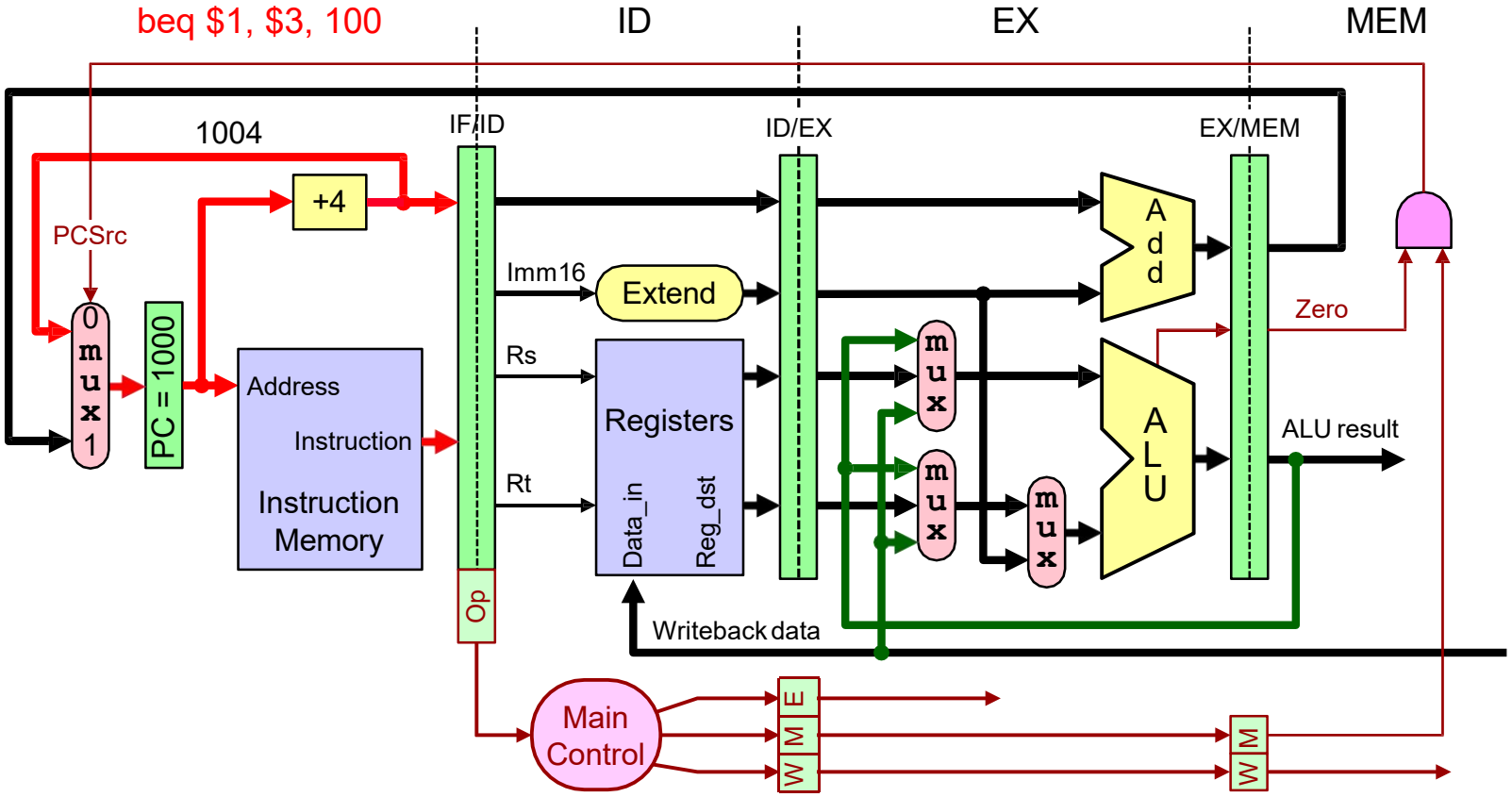
3-Cycle Branch Delay

- 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



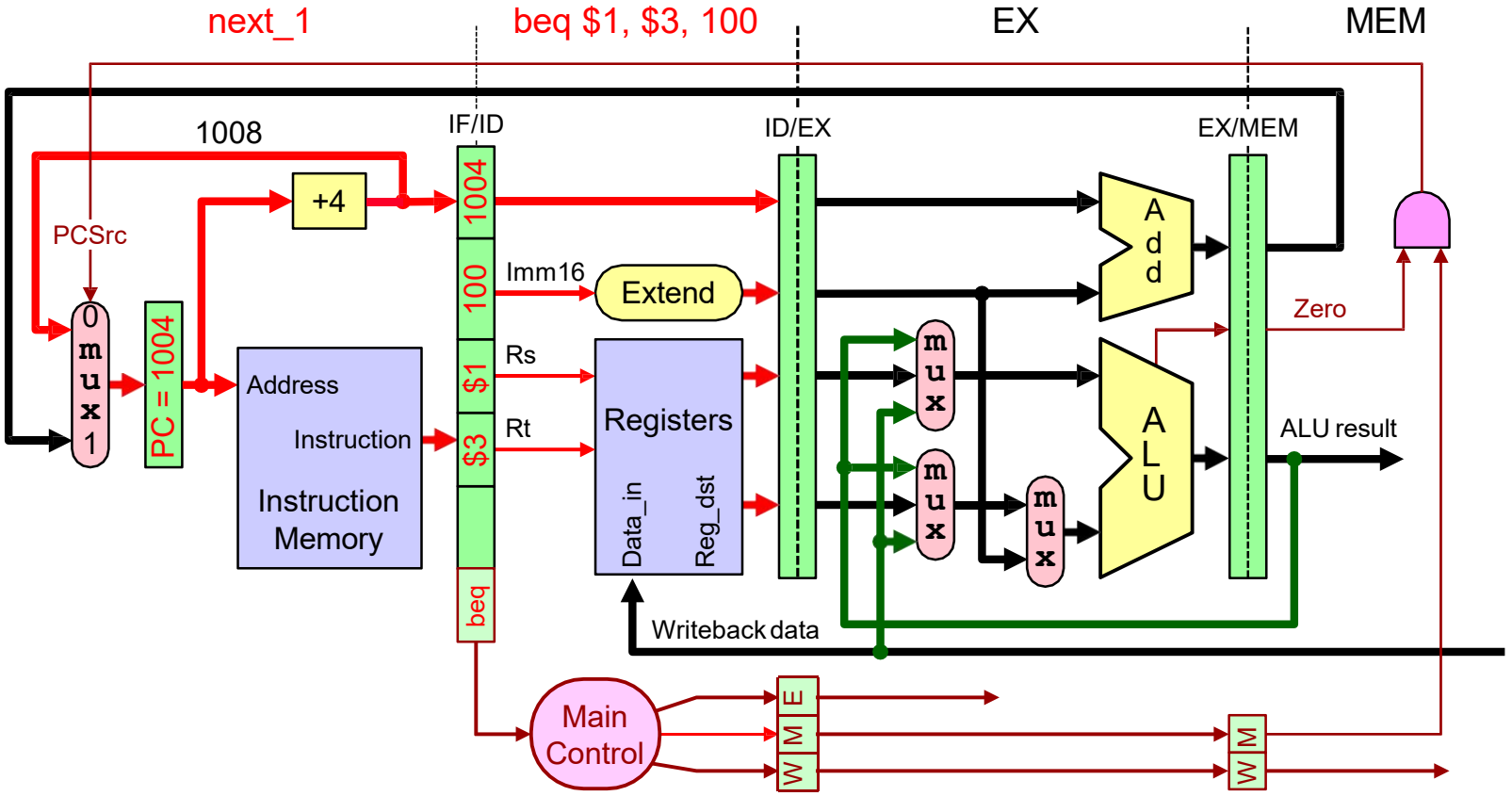
Branch Delay - CC1

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



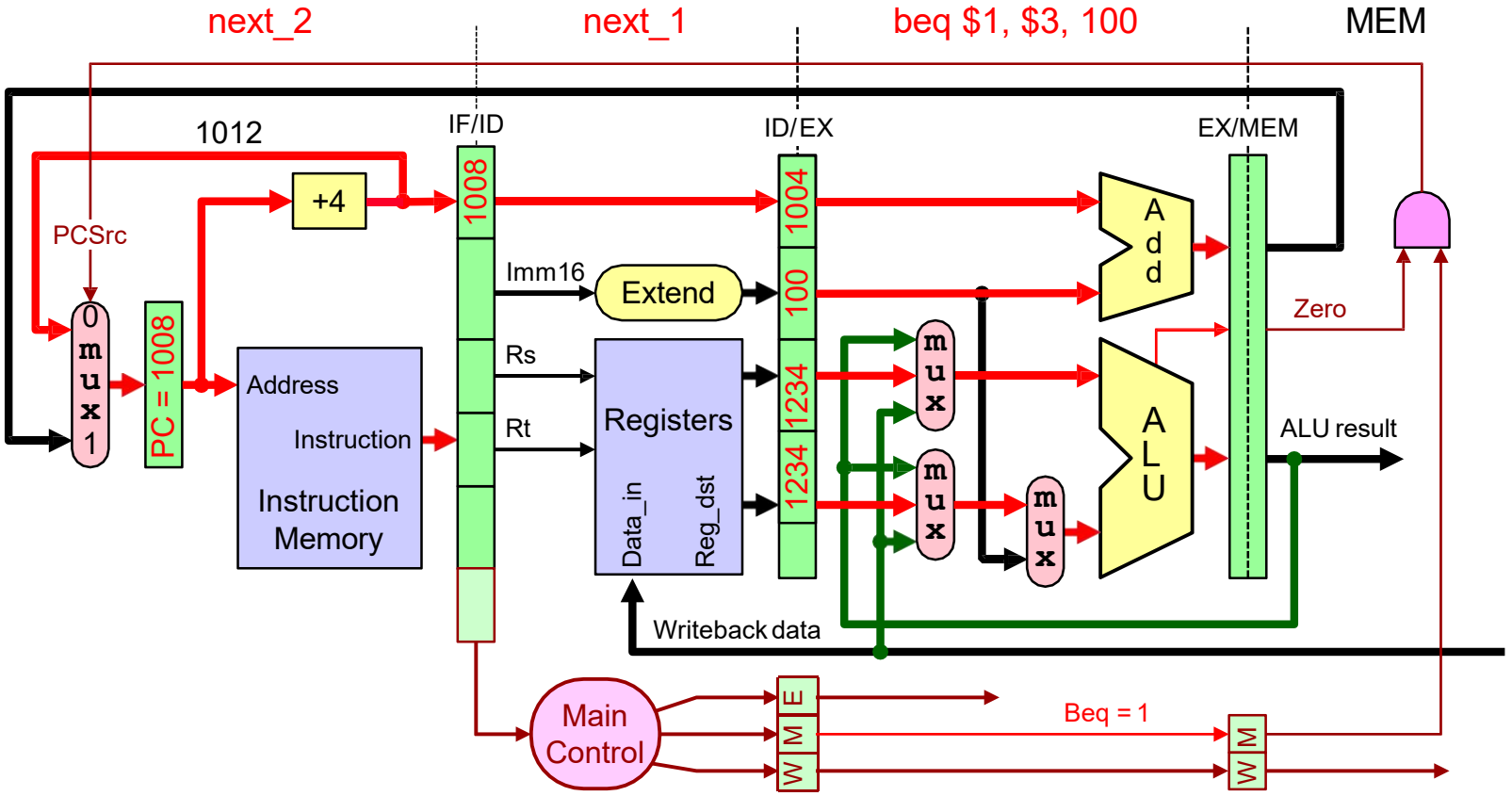
Branch Delay - CC2

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



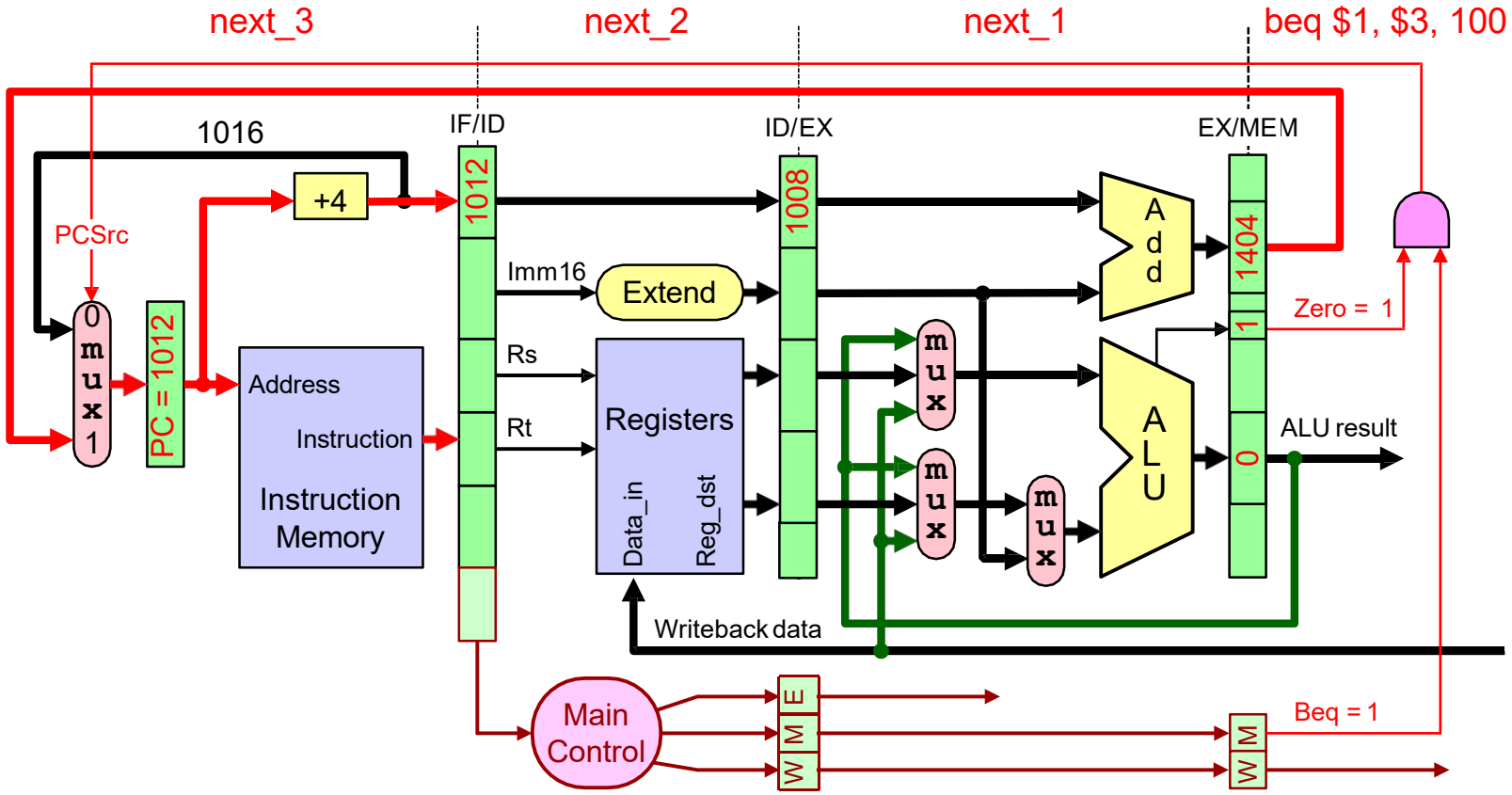
Branch Delay - CC3

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



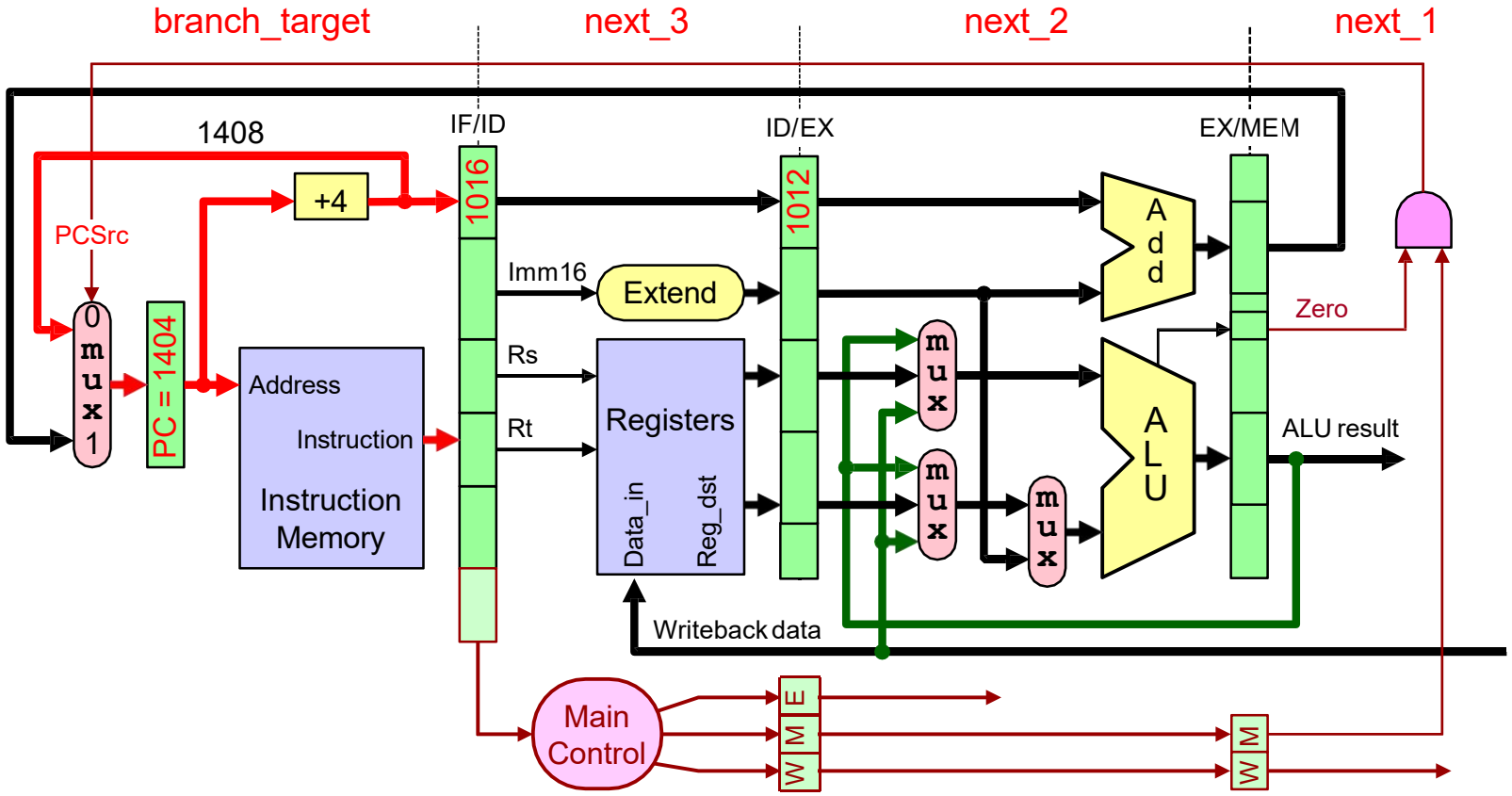
Branch Delay - CC4

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

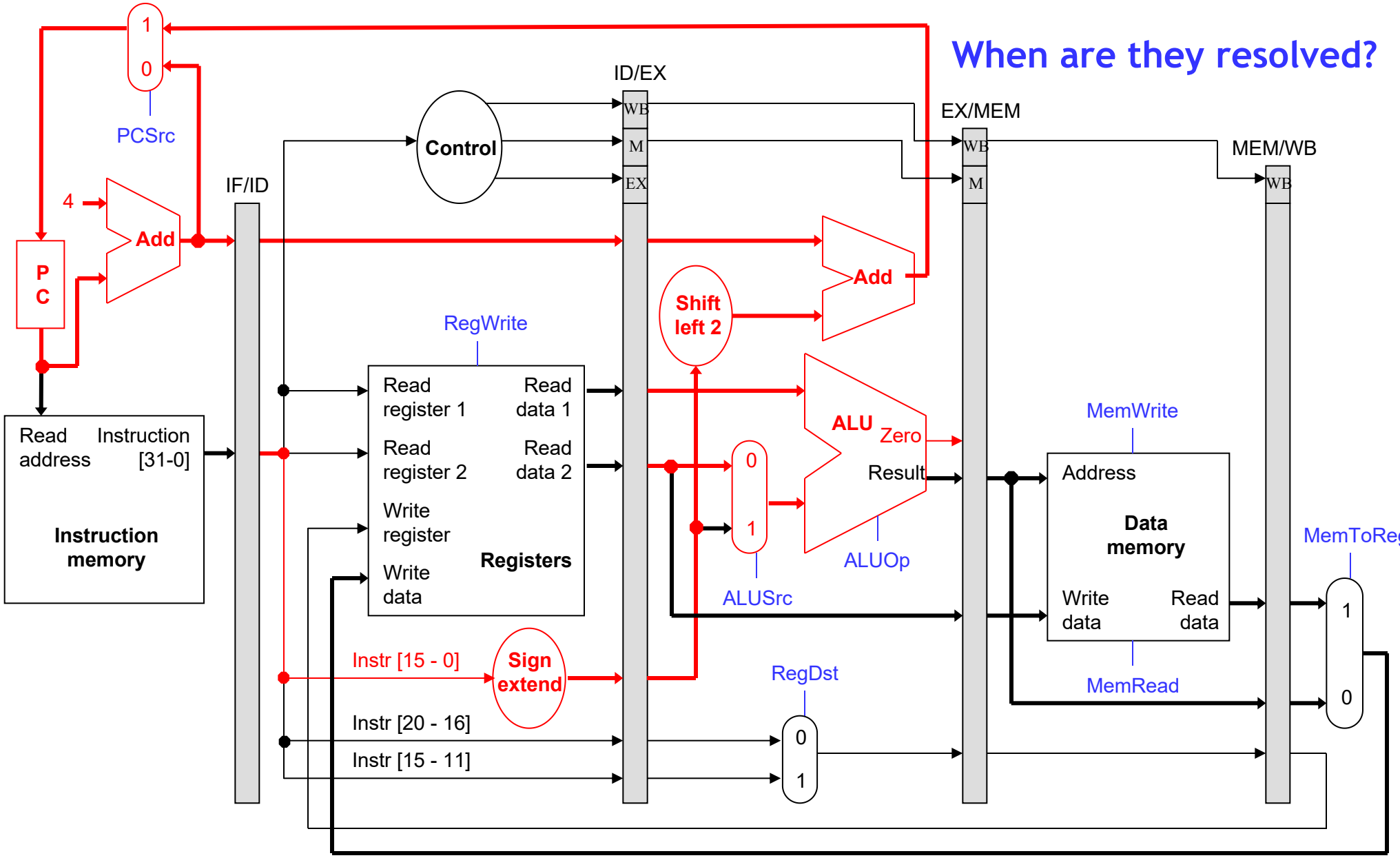


Branch Delay - CC5

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



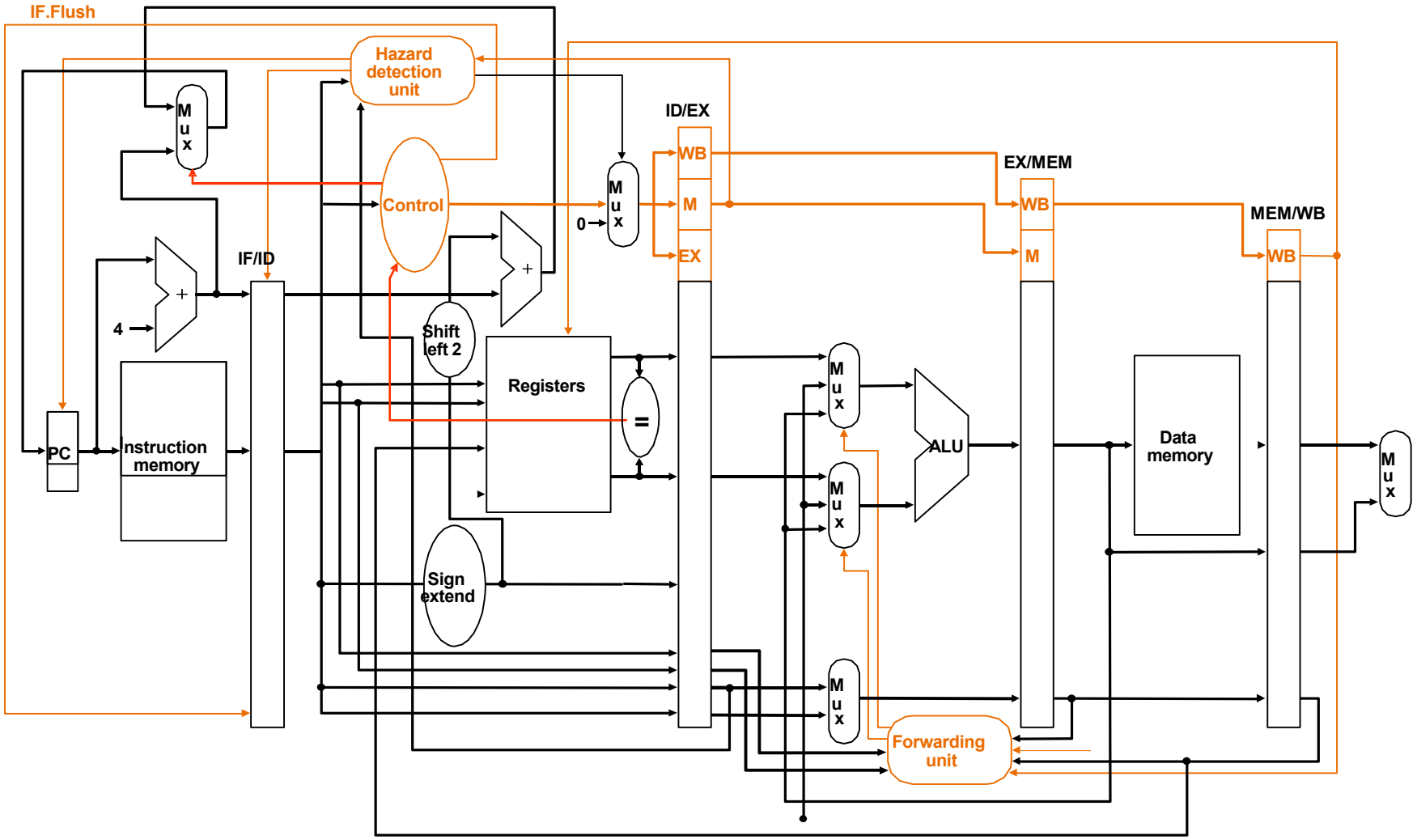
2-Cycle Branch Delay



Reducing the Delay of Branches from 2 cycle to 1 cycle

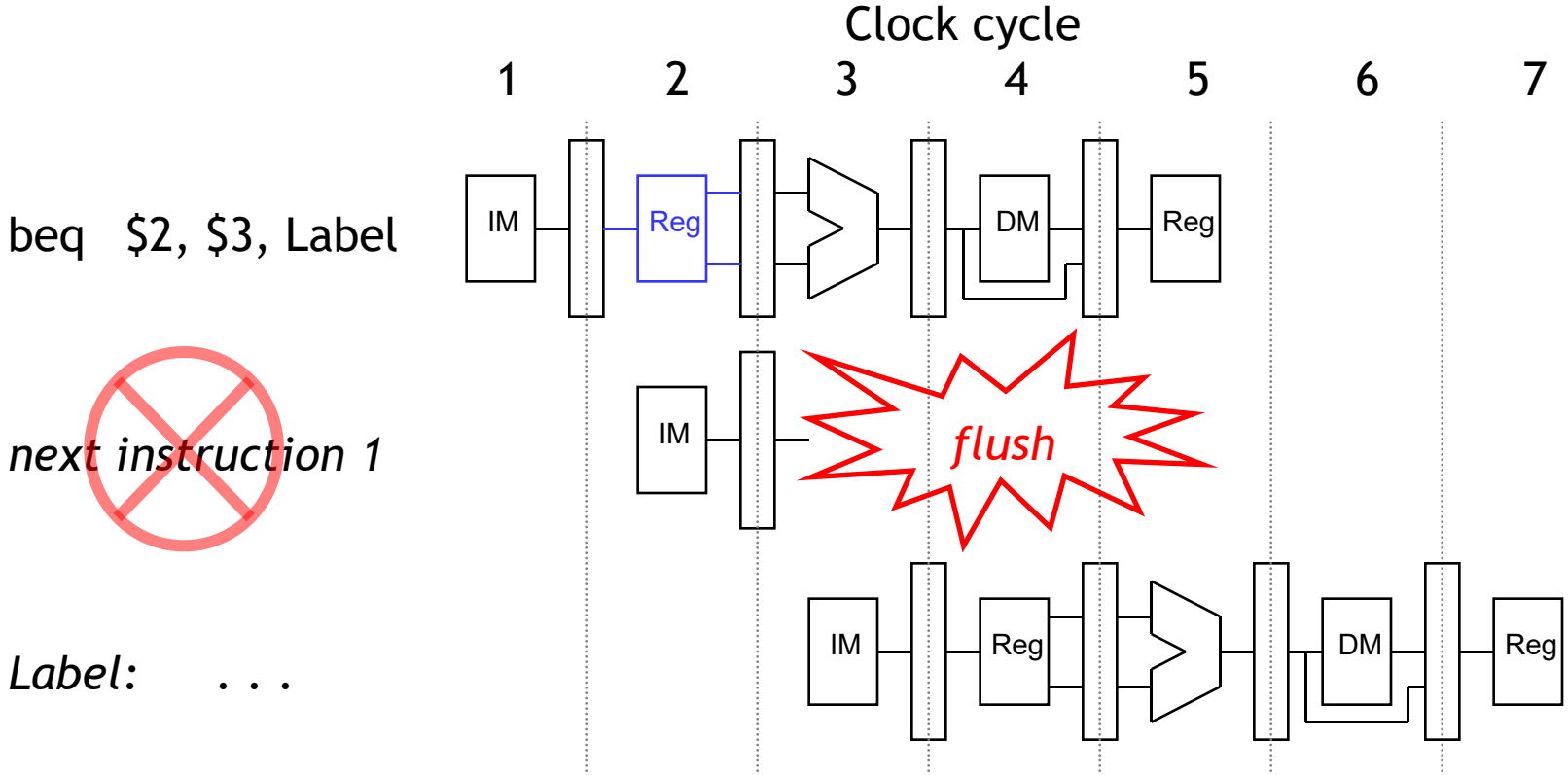
- Branch delay can be reduced from 3 cycles to just 1 cycle
- Branch decision is moved from 4th into 2nd pipeline stage
  - Branches can be determined earlier in the ID stage
  - Branch address calculation adder is moved to ID stage
  - A comparator in the ID stage to compare the two fetched registers
    - 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 to 1 cycle branch delay



Implementing branches(with 1 cycle branch delay)

- We can actually decide the branch a little earlier, in ID instead of EX.
  - Our sample instruction set has only a BEQ.
  - We can add a small comparison circuit to the ID stage, after the source registers are read.
- Then we would only need to flush one instruction on a misprediction.



Implementing flushes

- We must flush one instruction (in its IF stage) if the previous instruction is BEQ and its two source registers are equal.
- We can flush an instruction from the IF stage by replacing it in the IF/ID pipeline register with a harmless nop instruction.
  - MIPS uses sll $0, $0, 0 as the nop instruction.
  - This happens to have a binary encoding of all 0s: 0000 .... 0000.
- Flushing introduces a bubble into the pipeline, which represents the one-cycle delay in taking the branch.

Stalling pipeline

Stall pipeline(3 cycle branch delay)

- The simplest scheme to handle branches is to freeze or flush the pipeline, holding or deleting any instructions after the branch until the branch destination is known.
- Advantage: simple both to software and hardware
- The pipeline behavior looks like :

	1	2	3	4	5	6	7	8	9	10
Branch	IF	ID	EX	MEM	WB					
Branch successor		S	S	S	IF	ID	EX	MEM	WB	
Branch successor+1						IF	ID	EX	MEM	WB

Stall pipeline(3 cycle branch delay)

Stalling pipeline

Stall pipeline(2 cycle branch delay)

- The simplest scheme to handle branches is to freeze or flush the pipeline, holding or deleting any instructions after the branch until the branch destination is known.
- Advantage: simple both to software and hardware
- The pipeline behavior looks like :

	1	2	3	4	5	6	7	8	9	10
Branch	IF	ID	EX	MEM	WB					
Branch successor		S	S	IF	ID	EX	MEM	WB		
Branch successor+1					IF	ID	EX	MEM	WB	

Stall pipeline(2 cycle branch delay)

Stalling pipeline

Stall pipeline(1 cycle branch delay)

- The simplest scheme to handle branches is to freeze or flush the pipeline, holding or deleting any instructions after the branch until the branch destination is known.
- Advantage: simple both to software and hardware
- The pipeline behavior looks like :

	1	2	3	4	5	6	7	8
Branch	IF	ID	EX	MEM	WB			
Branch successor		S	IF	ID	EX	MEM	WB	
Branch successor+1				IF	ID	EX	MEM	WB

Stall pipeline(1 cycle branch delay)

Delayed branch

In a delayed branch, the execution cycle with a branch delay of length n is

Branch instr
sequential successor 1
sequential successor 2
. . . . .
sequential successor n
Branch target if taken

Sequential successors are in the branch-delay slots. These instructions are executed whether or not the branch is taken.

Delay Slot

- ISA says N instructions after branch/jump always executed
  - MIPS has 1 branch delay slot
  - i.e. Whether branch taken or not, instruction following branch is always executed

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**		

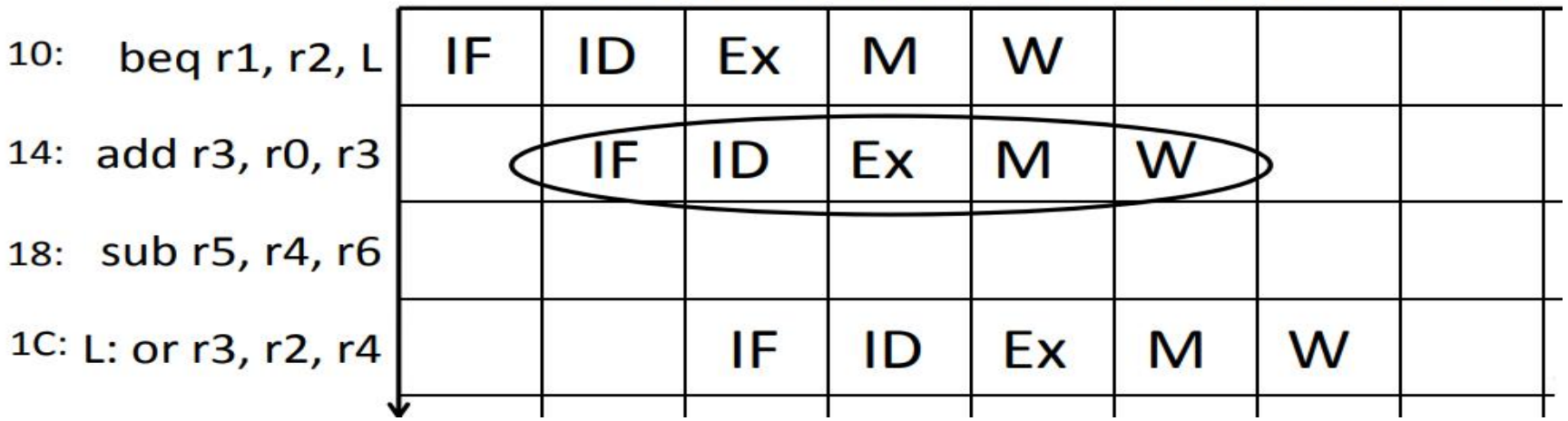
- Compiler/assembler fills the branch delay slot
  - By selecting a useful instruction

- **Delayed branches – code rearranged by compiler to place independent instruction after every branch (in delay slot).**

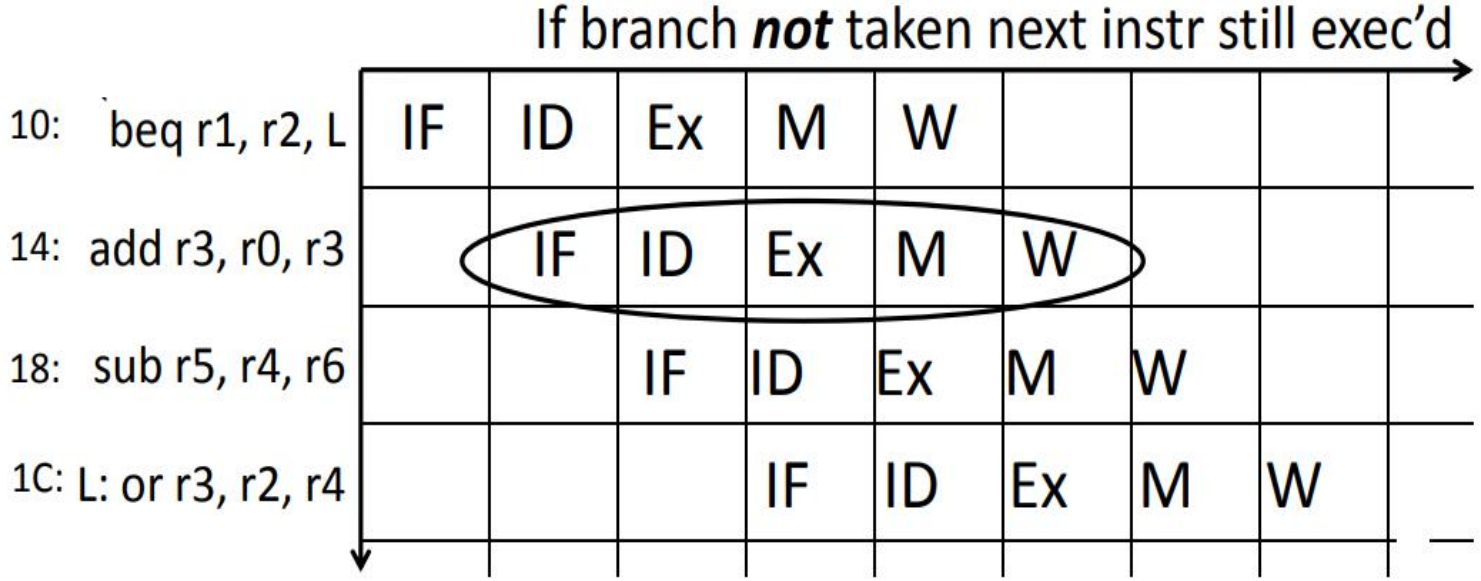
```
add $R4,$R5,$R6          beq $R1,$R2,20
beq $R1,$R2,20     ➡     add $R4,$R5,$R6
lw $R3,400($R0)          lw $R3,400($R0)
```

Delay slot

If branch ***taken*** next instr still exec'd



If branch ***not*** taken next instr still exec'd



Branch Prediction Schemes

- There are many methods to deal with the pipeline stalls caused by branch delay.
- compile-time schemes in which predictions are static - they are fixed for each branch during the entire execution, and the predictions are compile-time guesses.

- **Predict taken**
- **Predict not taken**
-

The pipeline with this scheme implemented behaves as shown below:

Untaken Branch Instr	IF	ID	EX	MEM	WB		
Instr i+1		IF	ID	EX	MEM	WB	
Instr i+2			IF	ID	EX	MEM	WB

Taken Branch Instr	IF	ID	EX	MEM	WB			
Instr i+1		IF	*idle*	*idle*	*idle*	*idle*		
Branch target			IF	ID	EX	MEM	WB	
Branch target+1				IF	ID	EX	MEM	WB

When branch is not taken, determined during ID, we have fetched the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall one clock cycle.

Timing

- If no prediction:

```
IF   ID   EX   MEM  WB
     IF   IF   ID     EX   MEM WB    --- lost 1 cycle
```

- If prediction:
  - If Correct

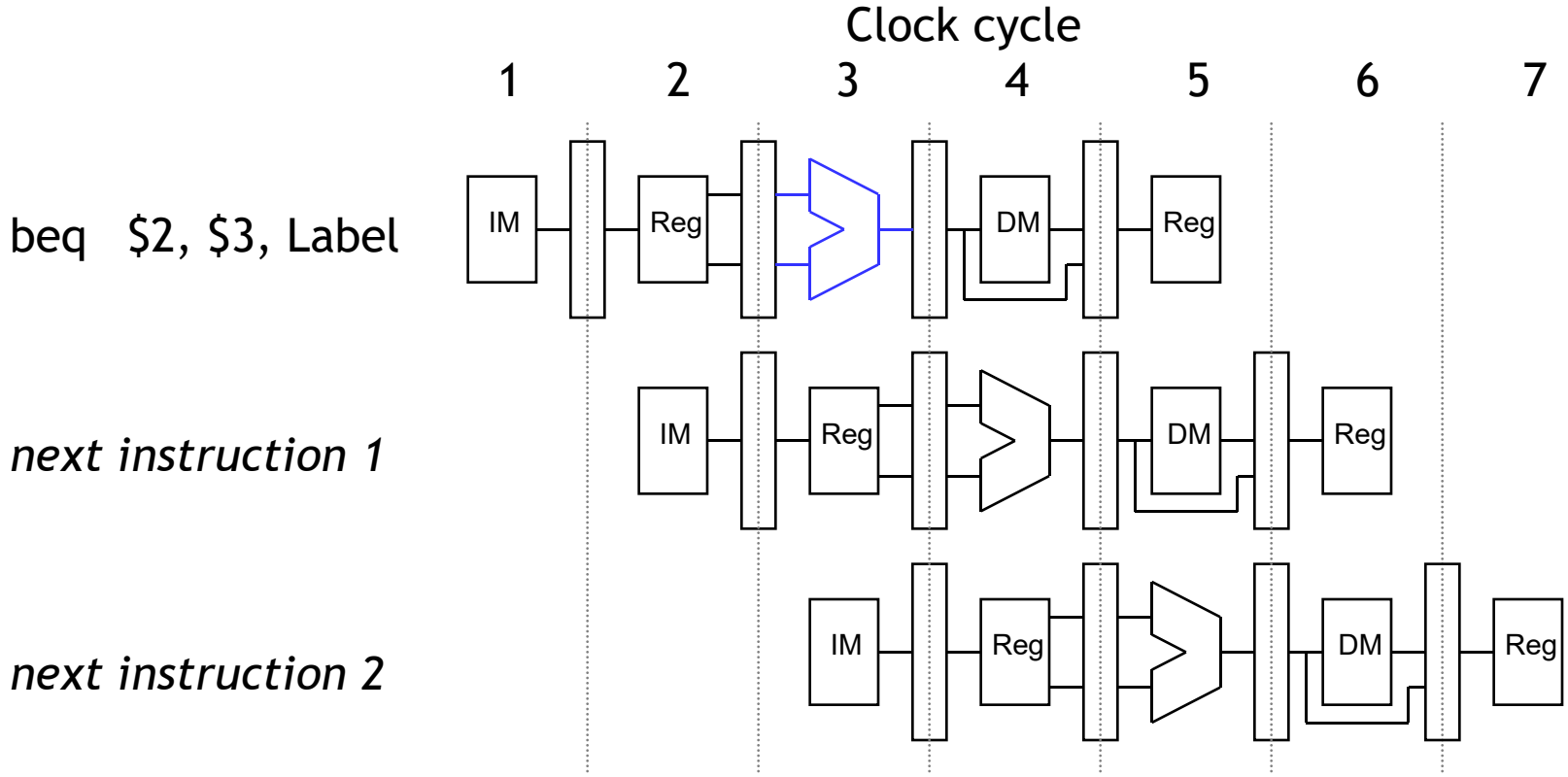
```
IF   ID  EX   MEM WB
     IF   ID   EX    MEM   WB    -- no cycle lost
```

  - If Misprediction:

```
IF   ID  EX   MEM  WB
     IF0  IF1  ID    EX   MEM  WB   --- 1 cycle lost
```

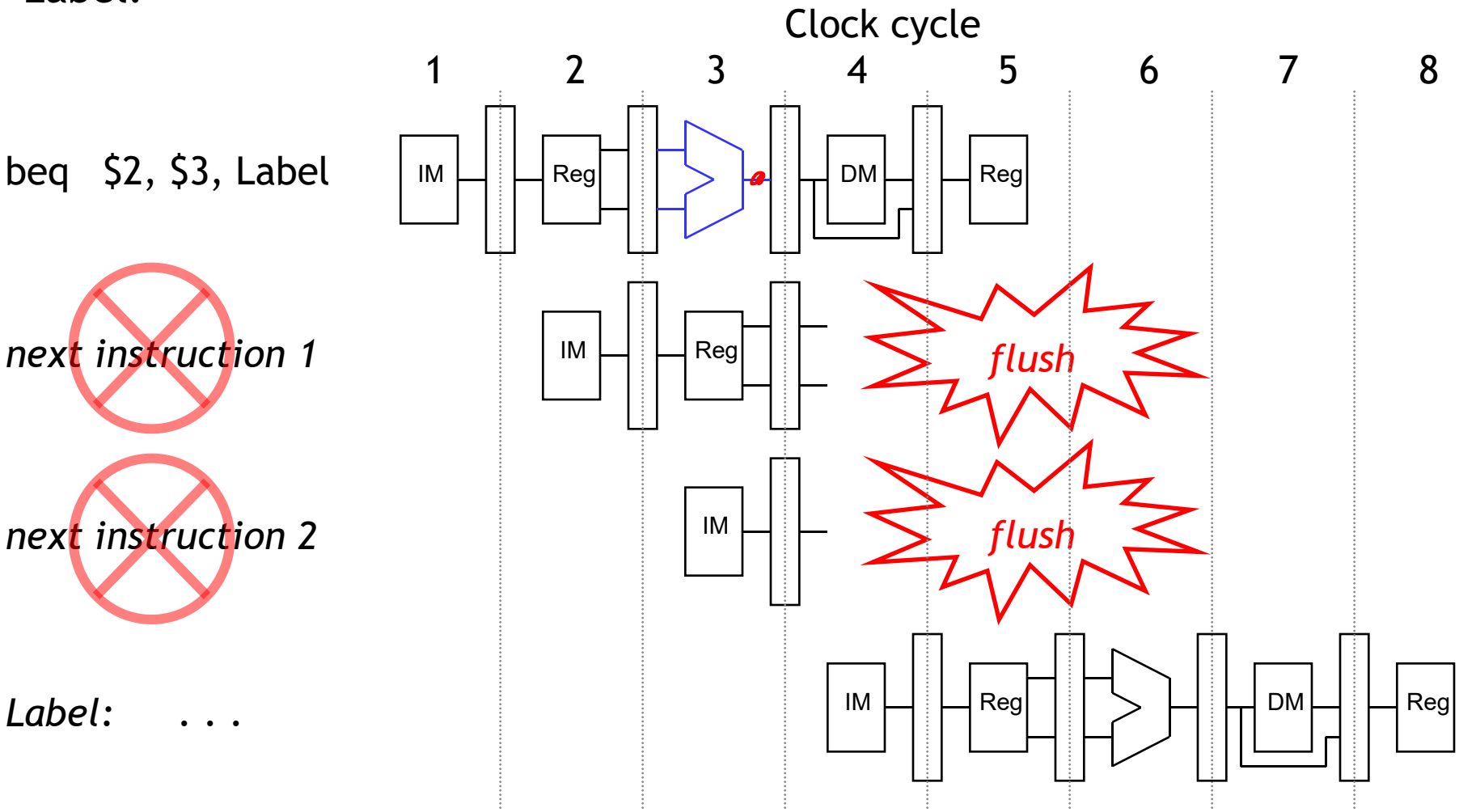
Branch prediction

- Another approach is to *guess* whether or not the branch is taken.
  - In terms of hardware, it's easier to assume the branch is *not* taken.
  - This way we just increment the PC and continue execution, as for normal instructions.
- If we're correct, then there is no problem and the pipeline keeps going at full speed.



Branch misprediction

- If our guess is wrong, then we would have already started executing two instructions incorrectly. We'll have to discard, or flush, those instructions and begin executing the right ones from the branch target address, Label.



Performance gains and losses

- Overall, branch prediction is worth it.
  - Mispredicting a branch means that clock cycles are wasted.
  - But if our predictions are even just occasionally correct, then this is preferable to stalling and wasting cycles for *every* branch.
- All modern CPUs use branch prediction.
  - Accurate predictions are important for optimal performance.
  - Most CPUs predict branches dynamically—statistics are kept at run-time to determine the likelihood of a branch being taken.
- The pipeline structure also has a big impact on branch prediction.
  - A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
  - We must also be careful that instructions do not modify registers or memory before they get flushed.

Scheduling the Branch Delay Slot with branch delay of 1 Clock cycle

- 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