Processor (Ch.4 part 4)

Dept. of AI·SW
Gachon University

Outline

- The Big picture
  - 4.1 Introduction
- Single-cycle design
  - 4.2 Logic Design Conventions – logic gates, components
  - 4.3 Building a Datapath
  - 4.4 A Simple Implementation Scheme
- **Multi-cycle design: Pipelining**
  - 4.5 An Overview of Pipelining
  - 4.6 Pipelined Datapath and Control
  - **4.7 Data Hazards: Forwarding versus stalling**
  - **4.8 Control Hazards**

Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy: why we have Instruction Memory and Data Memory
- **Data hazard**
  - Need to wait for previous instruction to complete its data read/write
- **Control hazard**
  - Deciding on control action depends on previous instruction

Data Hazards in ALU Instructions

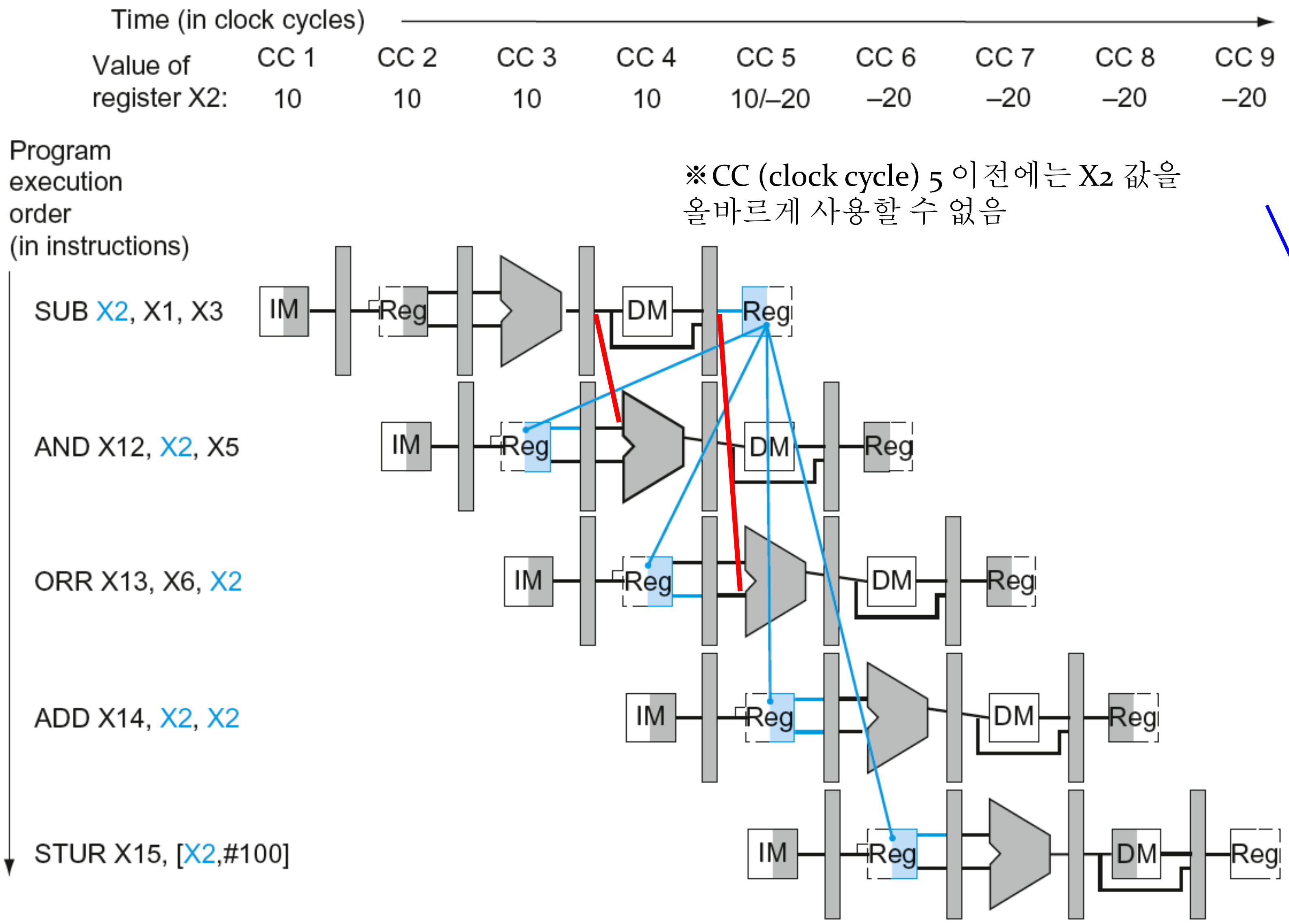
- Consider this sequence:

```
SUB  X2, X1,X3
AND  X12,X2,X5
OR   X13,X6,X2
ADD  X14,X2,X2
STUR X15,[X2,#100]
```

Dependency:
기록해야 할 register #와
읽어야할 register #가
같을 때..

- We can resolve hazards with forwarding
  - How do we detect when to forward?

Dependencies & Forwarding



※CC (clock cycle) 5 이전에는 X2 값을 올바르게 사용할 수 없음

:pipelined dependences

Forwarding

한CC에 같은 register에 W, R 같이 일어나면?
first half: Write
second half: Read
(가정함)
그래서 ADD 명령어 실행에선 hazard 안일어남

Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRn1, ID/EX.RegisterRm2
- Data hazards when
  - **EX hazard**

    1a. EX/MEM.RegisterRd = ID/EX.RegisterRn1

    1b. EX/MEM.RegisterRd = ID/EX.RegisterRm2
  - **MEM hazard**

    2a. MEM/WB.RegisterRd = ID/EX.RegisterRn1

    2b. MEM/WB.RegisterRd = ID/EX.RegisterRm2

Fwd from EX/MEM pipeline reg

계산 해야 하는데, 같은 레지스터를 사용중인 이전 instruction에서 이제 막 EX stage 에서 계산이 마쳤을 때,

Fwd from MEM/WB pipeline reg

계산 해야 하는데, 같은 레지스터를 사용중인 이전 instruction에서 이제 막 MEM stage에서 값을 read해 왔을 때,

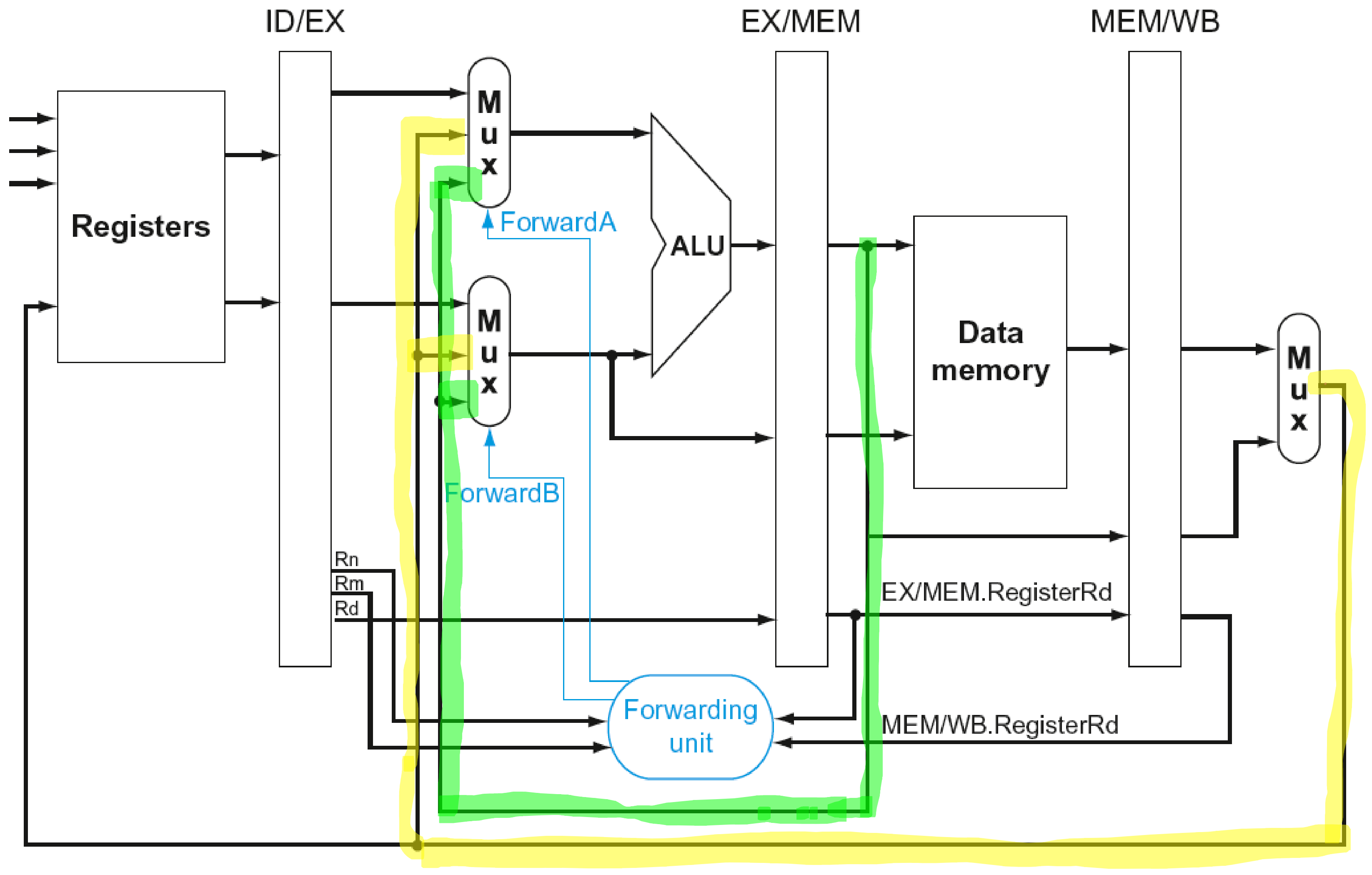
1a, 2a => forwardA   1b, 2b => forwardB

Detecting the Need to Forward

- But **only if** forwarding instruction will **write** to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite (<=asserted된 경우만)
- And only if Rd for that instruction is not XZR
  - EX/MEM.RegisterRd $\neq$ 31,
    MEM/WB.RegisterRd $\neq$ 31

앞 조건이 맞다고 무조건 할 필요 없이, RegWrite 신호가 활성화 되어 있을 때만..
XZR은 제외~

Forwarding Paths



Forwarding Conditions

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

Double Data Hazard

- Consider the sequence:

```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

What is "double" data hazard ?

2번째 X1, 1번째 실행 후 결과
3번째 X1은 2번째 실행 후 결과

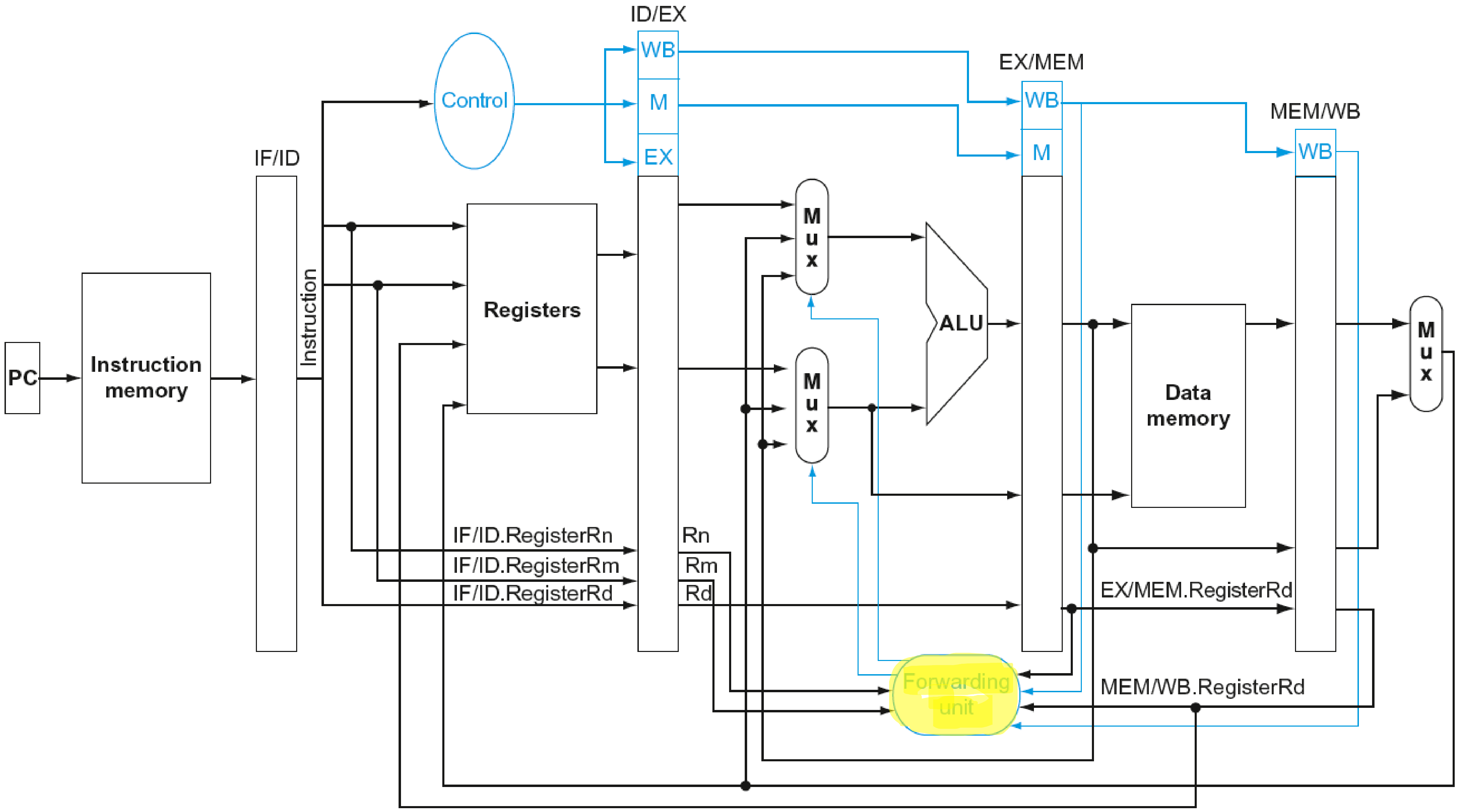
- Both hazards occur
  - Want to use **the most recent**
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true
    - (EX/MEM.RegisterRd == MEM/WB.RegisterRd == ID/EX.RegisterRn1 이면 double data hazard ➔ EX/MEM의 값을 forward 하도록 처리)

Revised Forwarding Condition

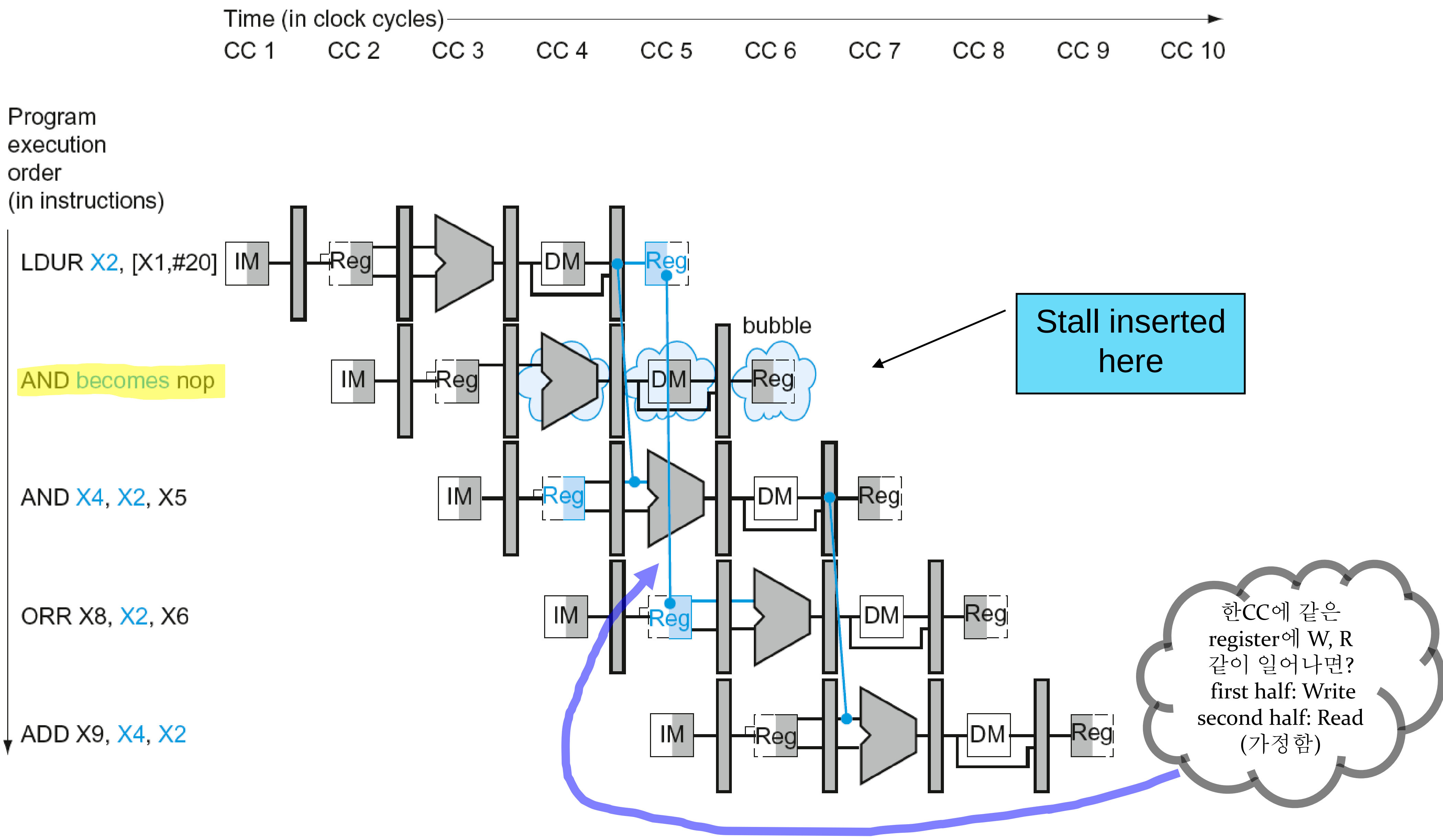
- MEM hazard
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 31)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRn1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) **ForwardA = 01**
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 31)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 31)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRm2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) **ForwardB = 01**

EX Hazard 아니면 (즉, double hazard 아니면), MEM Hazard로 판단하고, MEM/WB 것을 forwarding

Datapath with Forwarding



Load-Use Data Hazard : Data Hazard Requiring a Stall



Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
  - Let ***hazard detection unit*** do this
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRn1, IF/ID.RegisterRm2
- Load-use hazard when

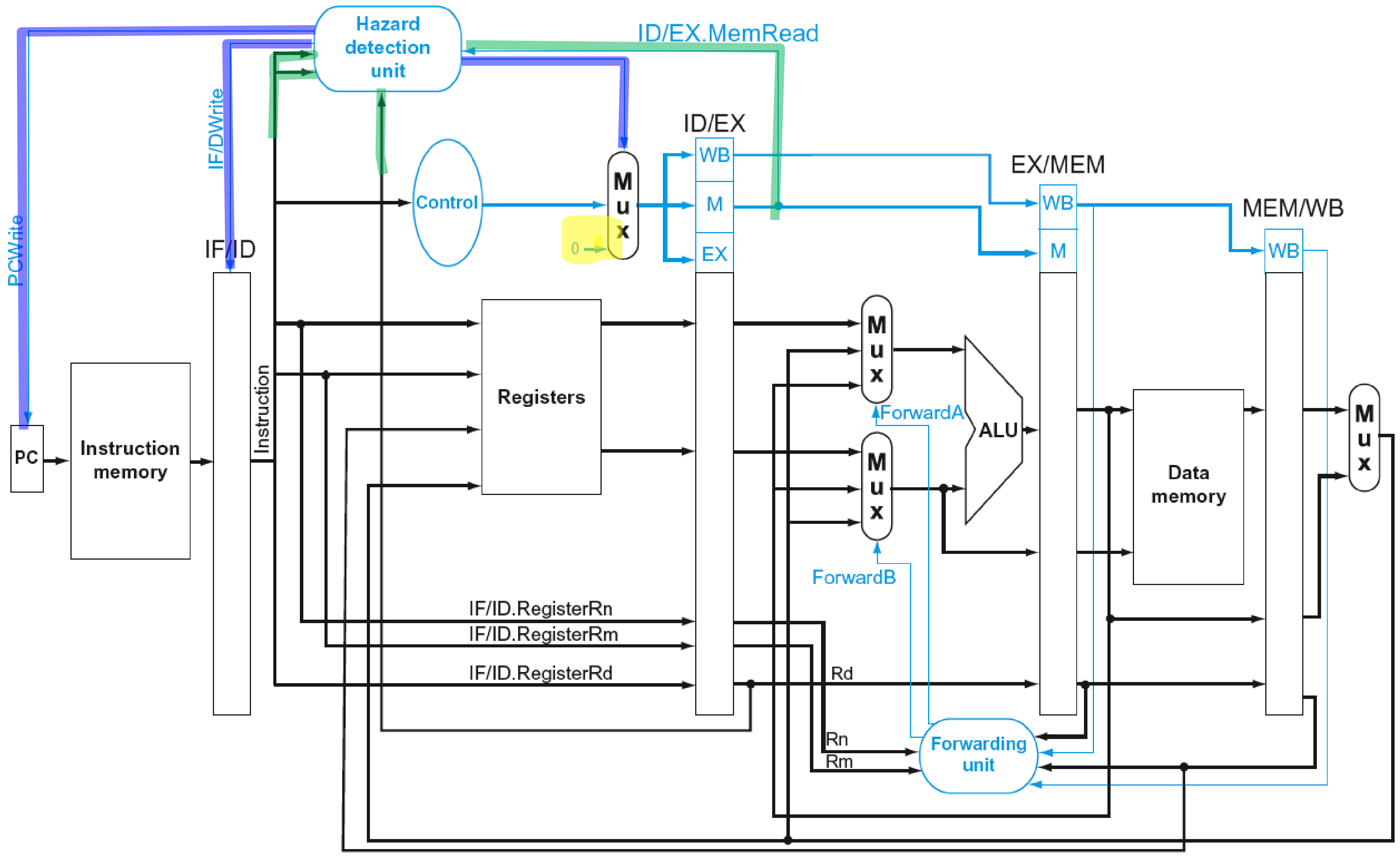
  -
    ```
    ID/EX.MemRead and
     ((ID/EX.RegisterRd = IF/ID.RegisterRn1) or
       (ID/EX.RegisterRd = IF/ID.RegisterRm1))
    ```
- If detected, stall and insert bubble

How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation : instruction that have no effect)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again

Datapath with Hazard Detection

nop 구현용



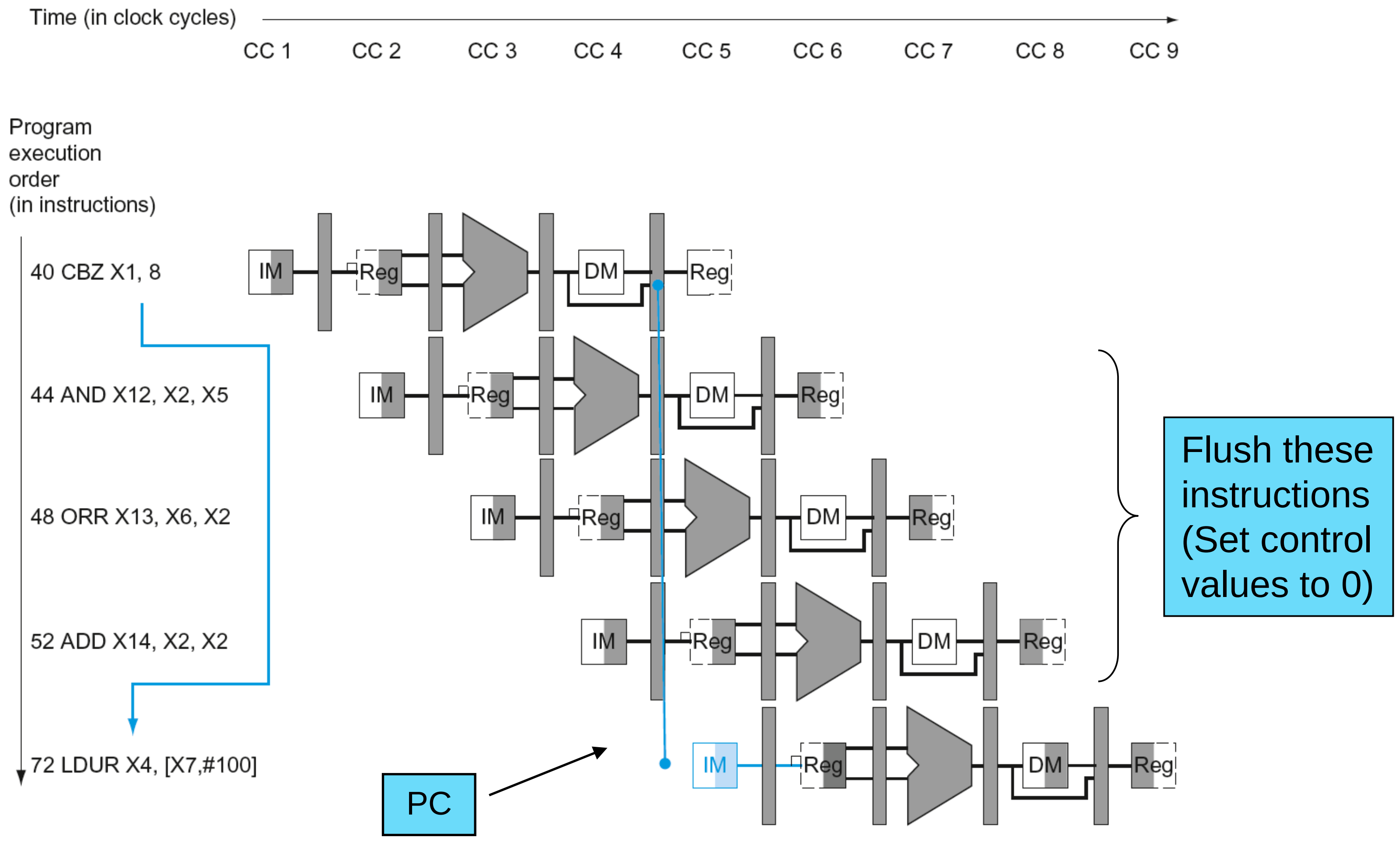
Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Control Hazards (Branch Hazards)

- If branch outcome determined in MEM



Reducing Branch Delay

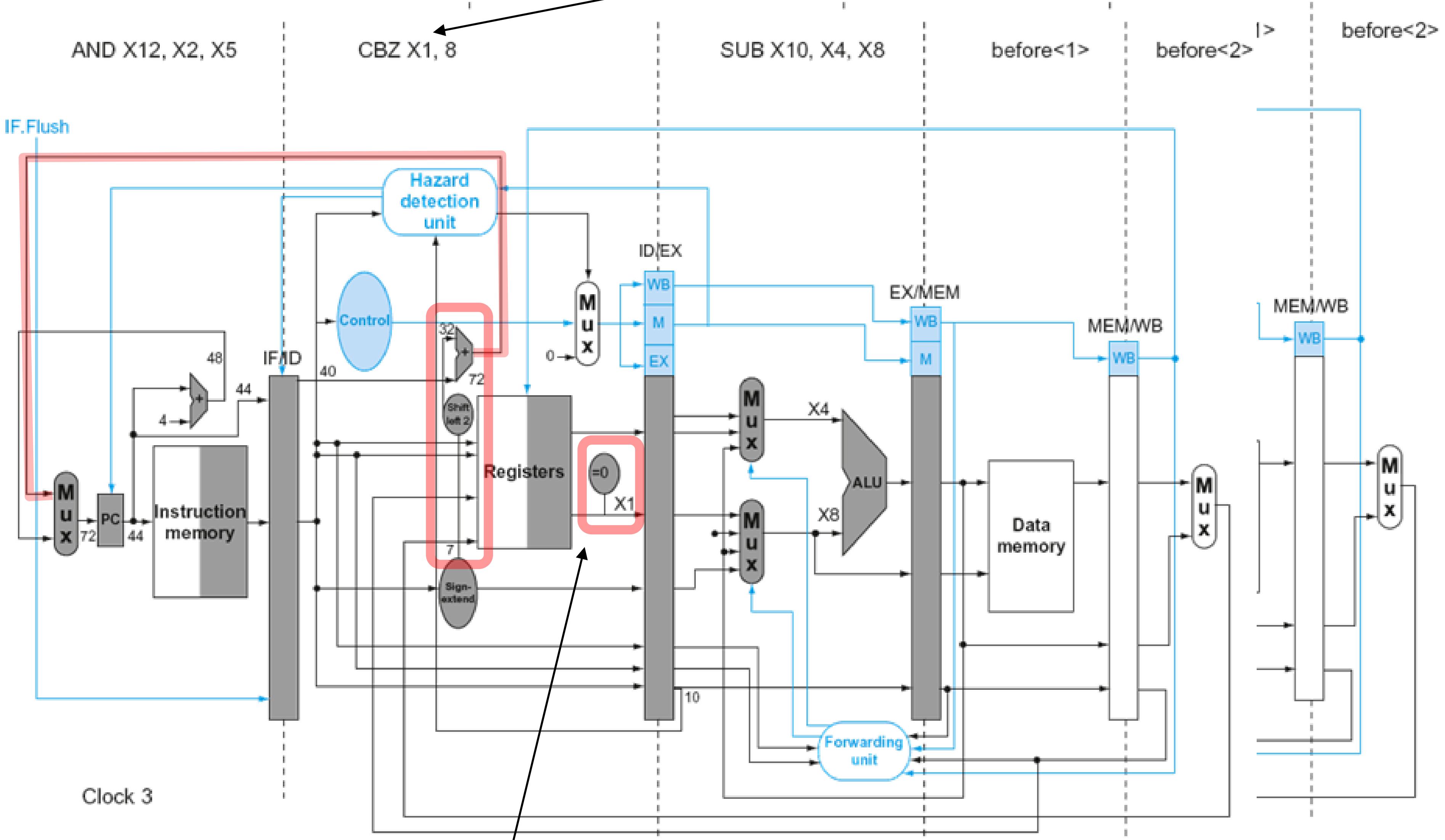
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:   SUB  X10, X4, X8
40:   CBZ  X1,  X3, 8
44:   AND  X12, X2, X5
48:   ORR  X13, X2, X6
52:   ADD  X14, X4, X2
56:   SUB  X15, X6, X7
      ...
72:   LDUR X4, [X7,#50]
```

Calculate target Addr @ID stage

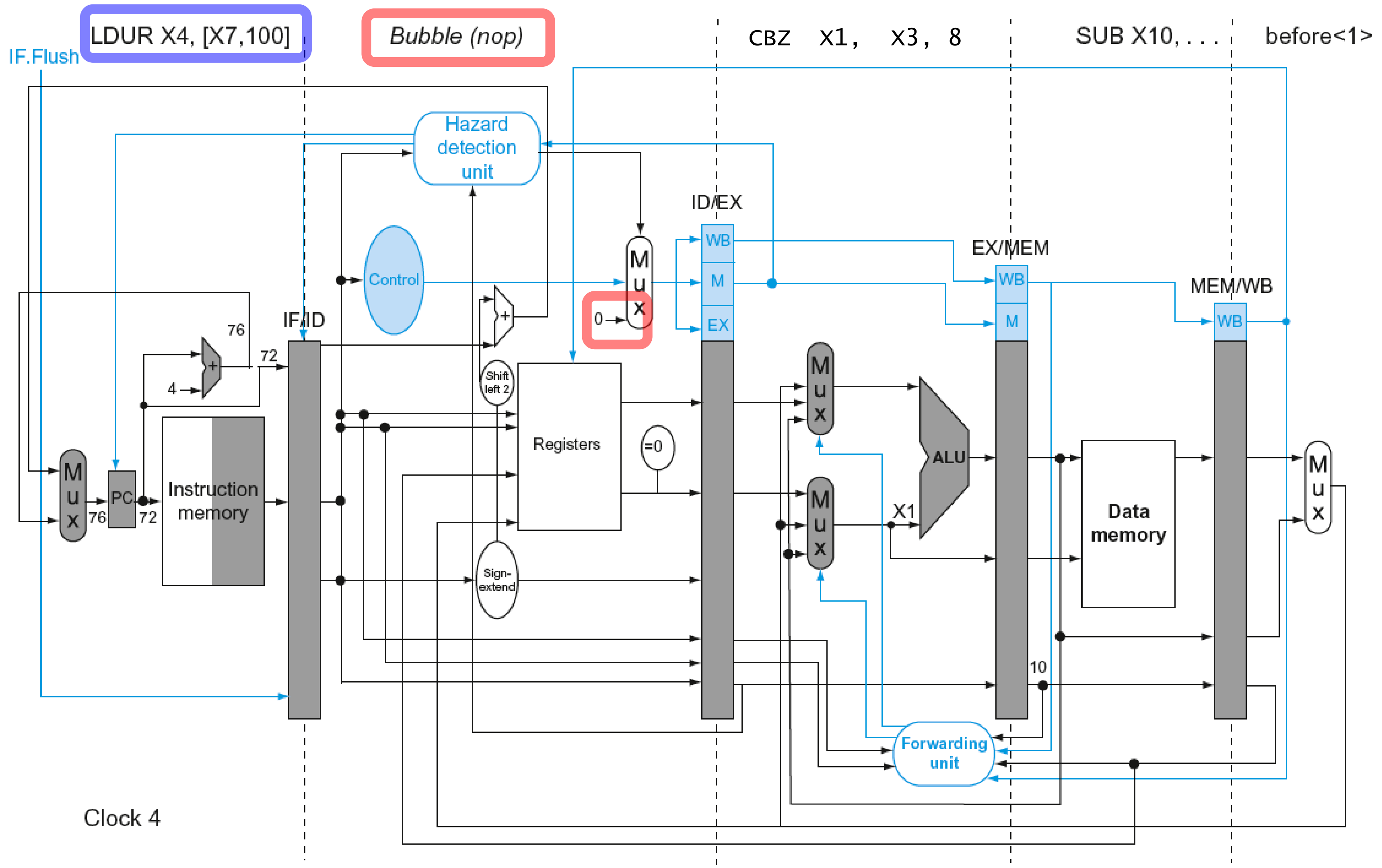
Example: Branch Taken

If X1= 0



Simply done by using OR operation

Example: Branch Taken



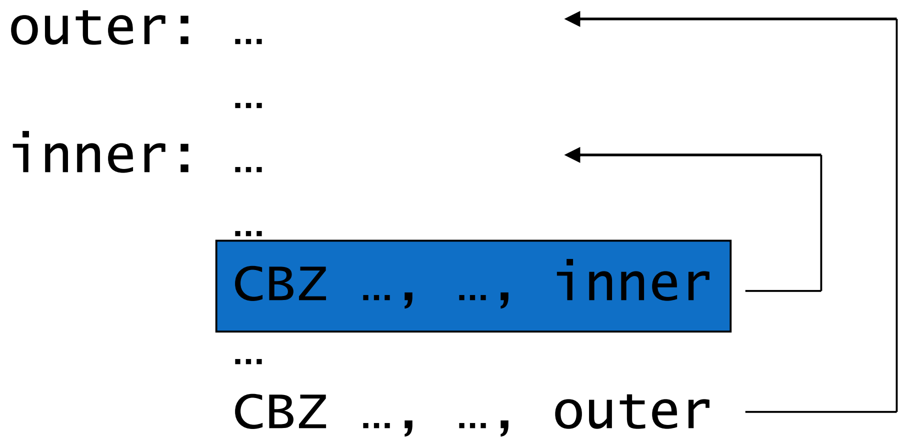
Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

- 이전에 뛰었으면 다음에도 뛰자
  → 1-Bit Predictor
- 이전에 두 번 뛰었으면 다음에도 뛰자
  → 2-Bit Predictor

1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

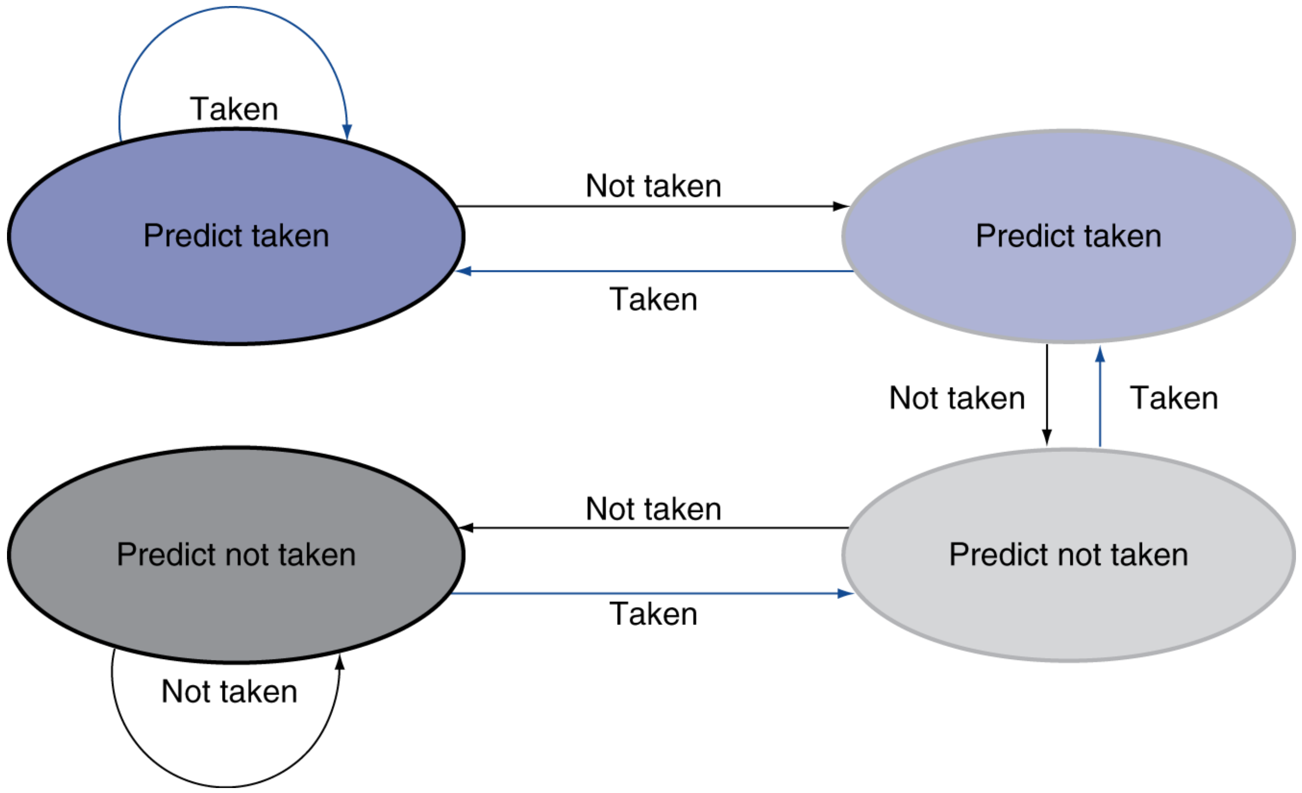


```
outer: …
       …
inner: …
       …
       CBZ …, …, inner
       …
       CBZ …, …, outer
```

  - 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

Branch target address를 IF 단계에서 계산해서 버퍼링하고, branch taken으로 prediction되면, 바로 branching