

## Topic 3

## Fundamental Processors II

**Xinfei Guo**  
**[xinfei.guo@sjtu.edu.cn](mailto:xinfei.guo@sjtu.edu.cn)**

**May 23<sup>rd</sup>, 2022**




# T3 learning goals

---

- Fundamental Processors (review)
  - Section I: Pipelining
  - **Section II: Hazards**

# What is hazard?

- Dependence: relationship between two instructions – Not a bad thing
- Hazards 
  - A situation that prevents starting the next instruction in the next clock cycle
  - Correctness problems associated w/ processor design
  - Hazards are a bad thing: stalls reduce performance (recall CPI) or complicate the processor design

# Hazards

- Structural hazards
  - Same resource needed for different purposes at the same time (Possible: ALU, Register File, Memory)
  - Why do we have IM and DM separately?
- Data hazards (Both memory and register)
  - Instruction output needed before it's available
  - Need to wait for previous instruction to complete its data read/write
- Control hazards (Procedural Dependence)
  - Flow of execution depends on previous instruction
  - One instruction affects whether another executes at all

# A quick quiz

Which of the following statements is true?

- A. Whether there is a data **dependence** between two instructions depends on the machine the program is running on.
- B. Whether there is a data **hazard** between two instructions depends on the machine the program is running on.
- C. Both A & B
- D. Neither A nor B

# Types of Data hazards

- Consider executing a sequence of register-register instructions of type:

Data dependence

$x3 \leftarrow x1 \text{ op } x2$       **Read-after-Write (RAW)**  
...  
 $x5 \leftarrow x3 \text{ op } x4$

Anti-dependence

$x3 \leftarrow x1 \text{ op } x2$       **Write-after-Read (WAR)**  
...  
 $x1 \leftarrow x4 \text{ op } x5$

Output-dependence

$x3 \leftarrow x1 \text{ op } x2$       **Write-after-Write (WAW)**  
...  
 $x3 \leftarrow x6 \text{ op } x7$

Note that the RAR (read after read) case is not a hazard.

# The Cost of Deeper Pipelines

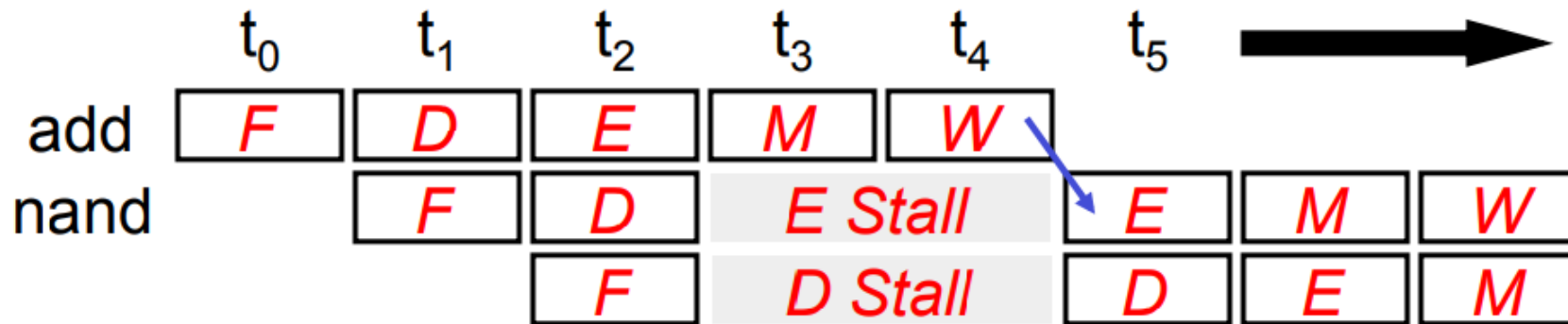
- Instruction pipelines are not ideal

- Assume

add 3 1 2

nand 5 3 4

*RAW!!*



# Terminology

- Hazard Resolution

- Static Method: Performed at compiled Time in software
- Dynamic Method: Performed at run Time using hardware

- **Interlock**

- Hardware mechanisms for dynamic hazard resolution
- Wait for hazard to clear by holding dependent instructions in issue stage



# Handling Data Hazards

- Avoidance (static)
  - Make sure that there are no hazards in the code
- Interlock/stall (dynamic)
  - Stall until earlier instructions finish
- forward/bypass (dynamic)
  - Resolve hazard earlier by bypassing value as soon as available
- Second and third require detection

# Avoidance (static)

- Programmer/compiler must know implementation details
  - Insert nops/bubbles between dependent instructions

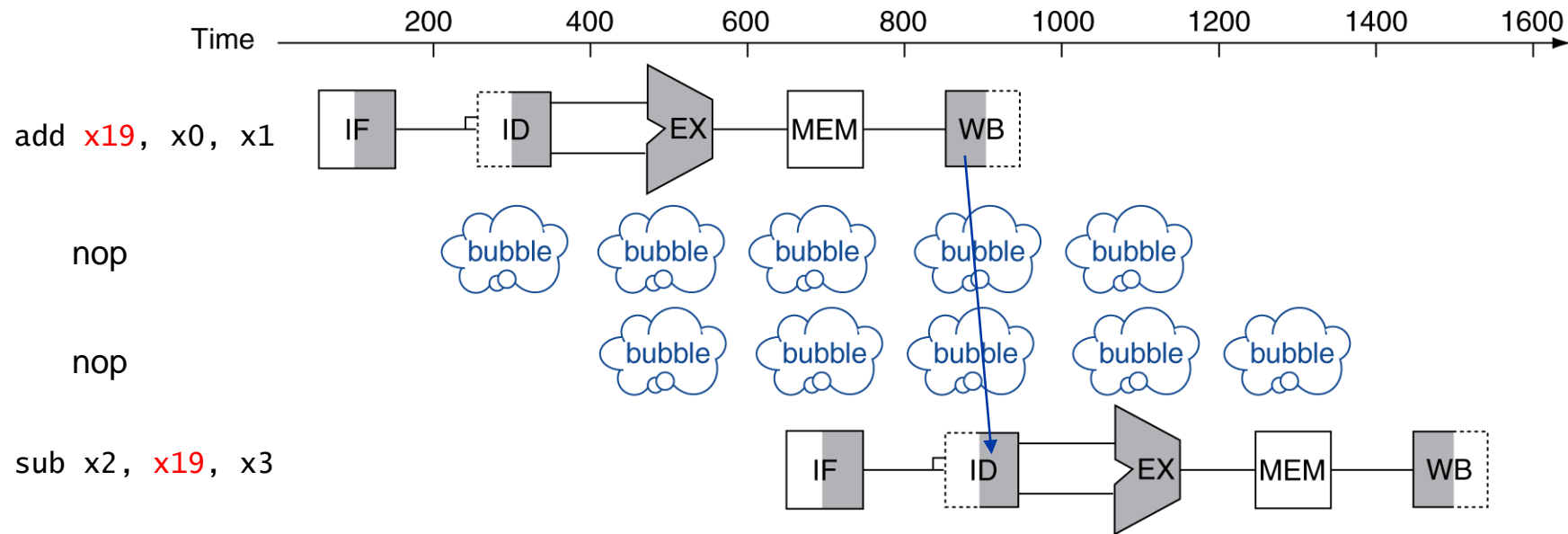
```
add    1  2  3    ← write R3 in cycle 5
nop
nop
nand   3  4  5    ← read R3 in cycle 6
```

# Any problems?

- Code size
  - Higher instruction cache footprint
  - Longer binary load times
  - Worse in machines that execute multiple instructions / cycle
    - Intel Itanium – 25-40% of instructions are nops
- Slower execution
  - CPI=1, but many instructions are nops

# Interlock/Stall (dynamic)

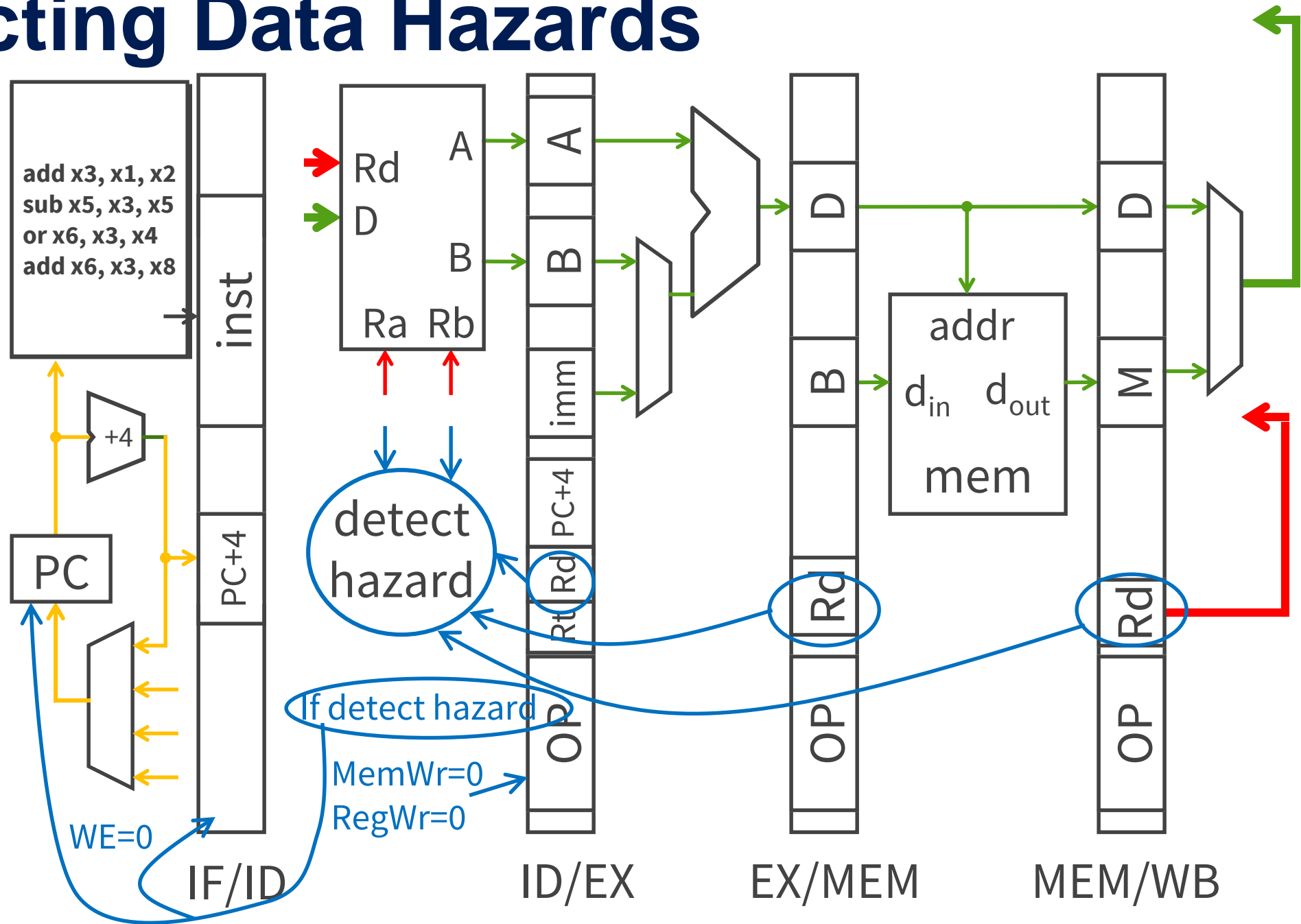
- Pause current and subsequent instructions till safe
  - By inserting **bubbles** or **stalls**  
nop    # no operation  
      # machine code: 0x00000000



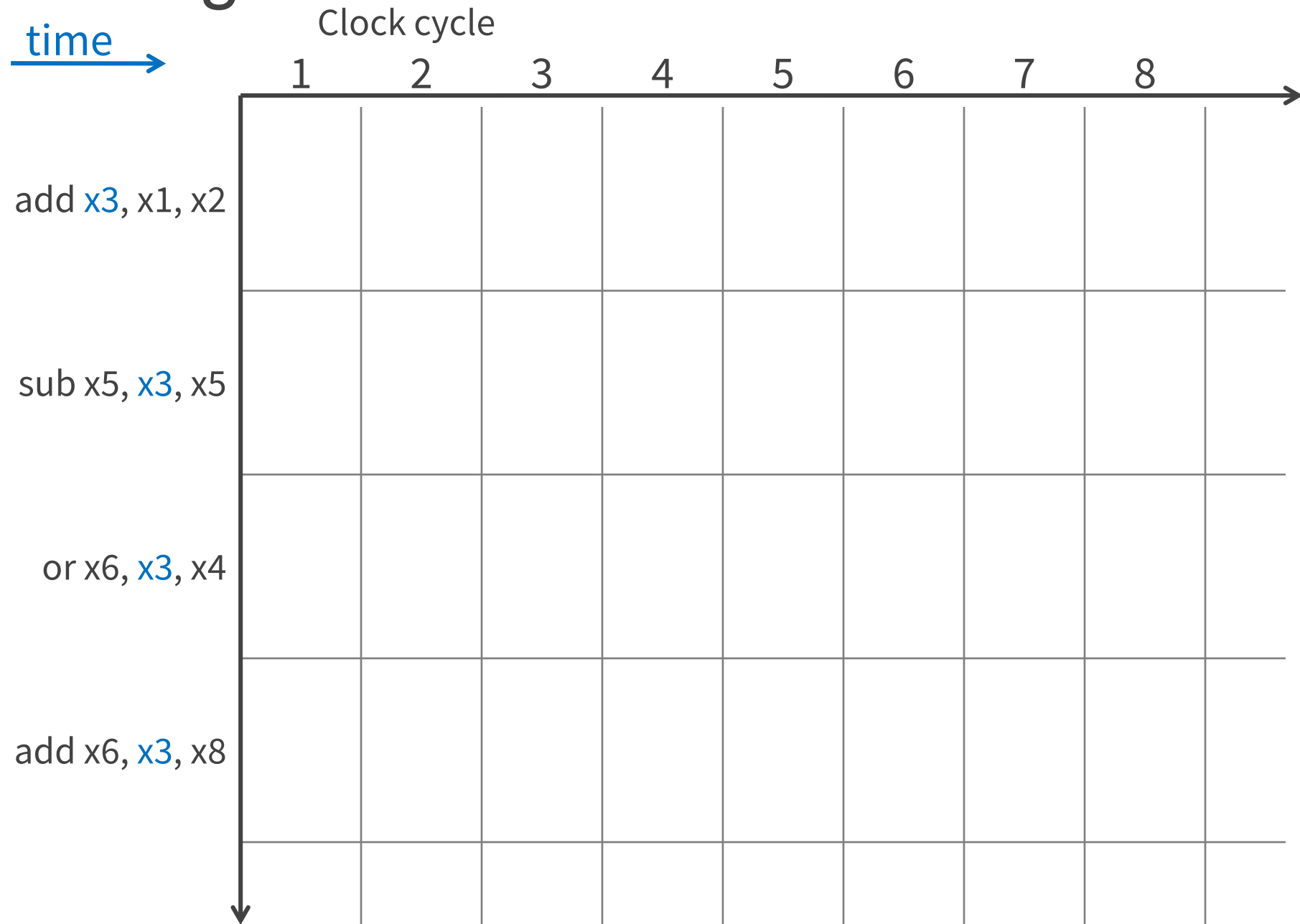
# Interlock/Stall

- How to stall an instruction in ID stage
  - prevent IF/ID pipeline register update
    - stalls the ID stage instruction
  - convert ID stage instr into **nop** for later stages
    - innocuous “bubble” passes through pipeline
  - prevent PC update
    - stalls the next (IF stage) instruction
- How to detect?
  - Compare regA & regB with DestReg of preceding insn
    - It is a 3 bit comparator

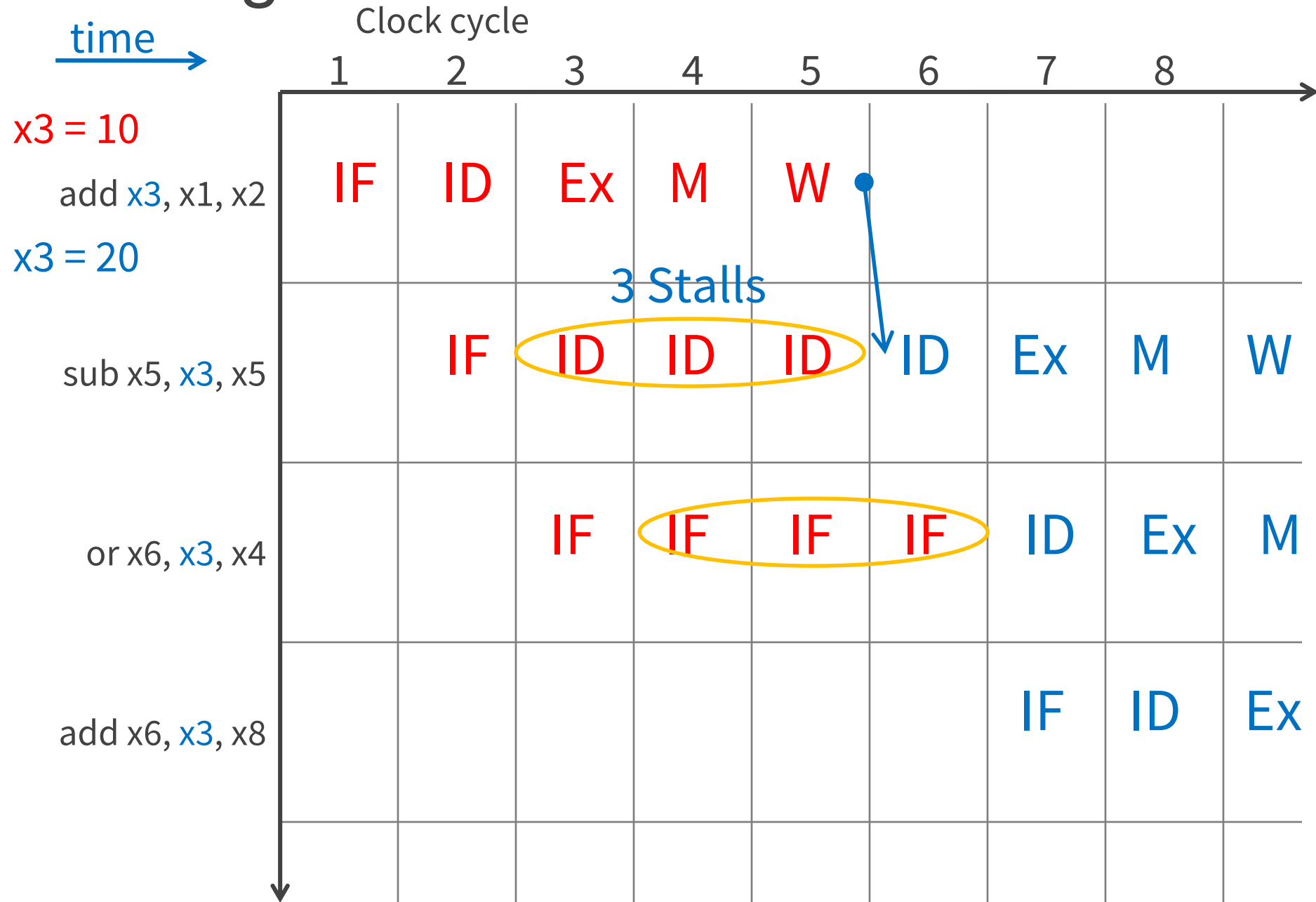
# Detecting Data Hazards



# Stalling

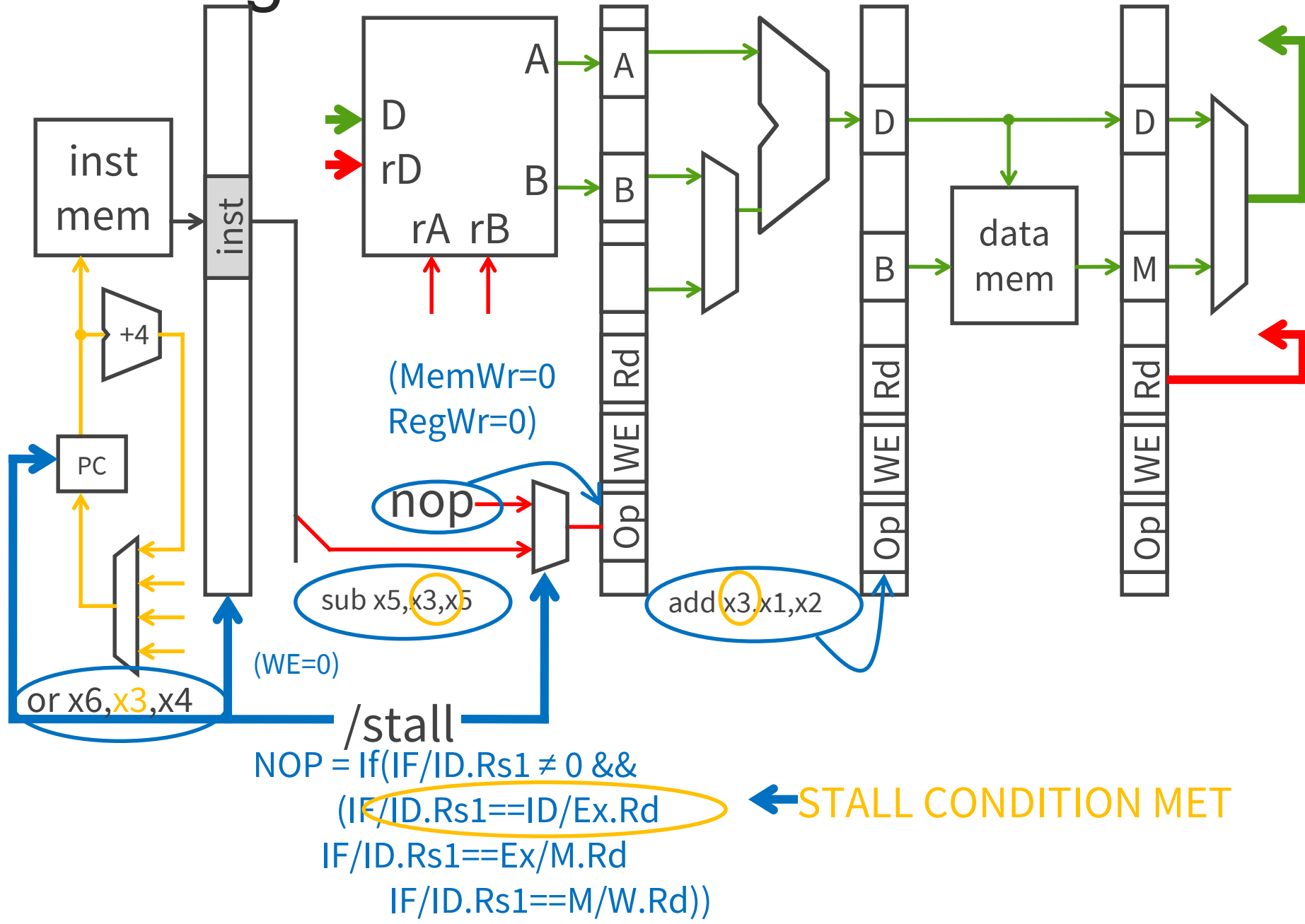


# Stalling

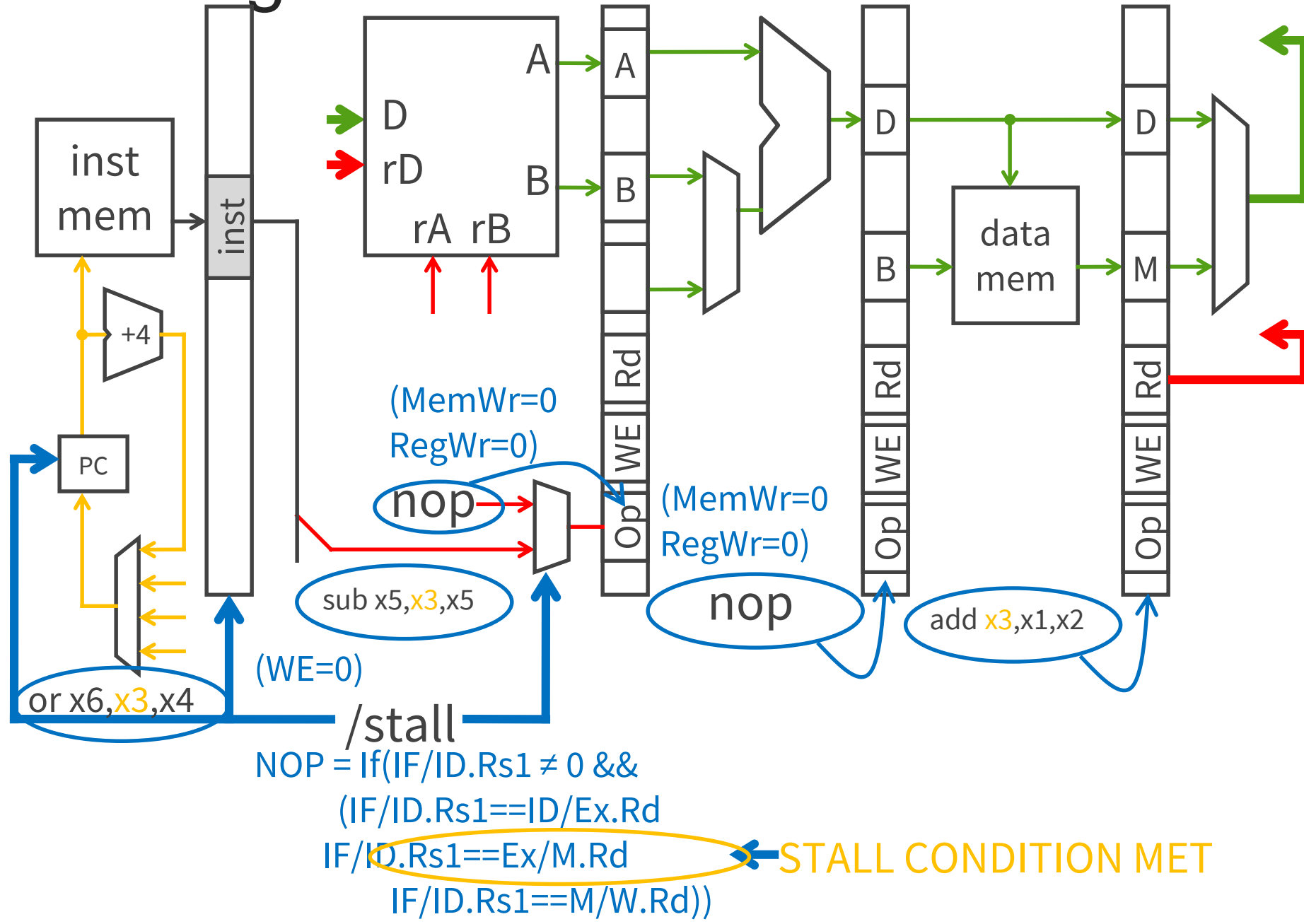




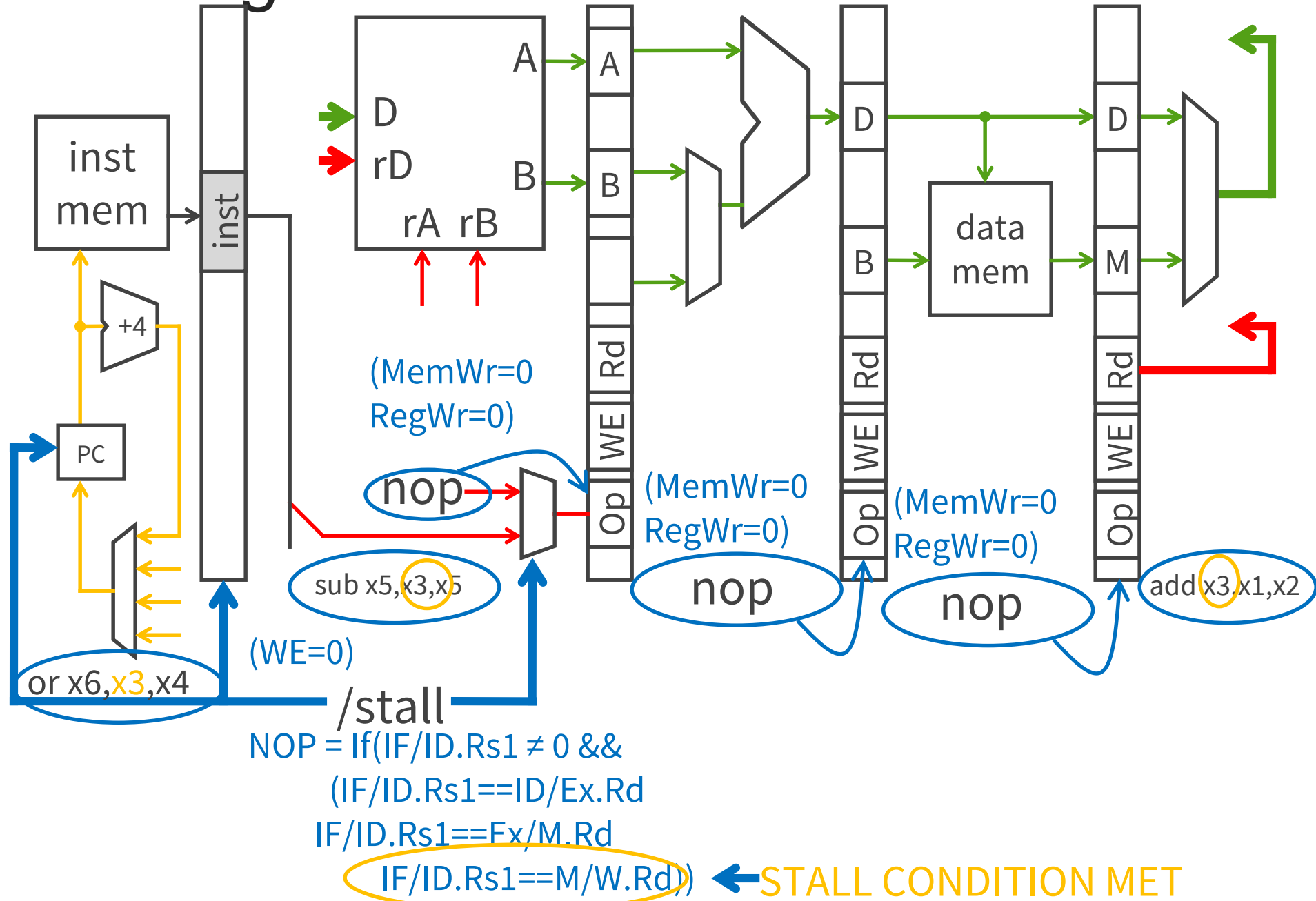
# Stalling



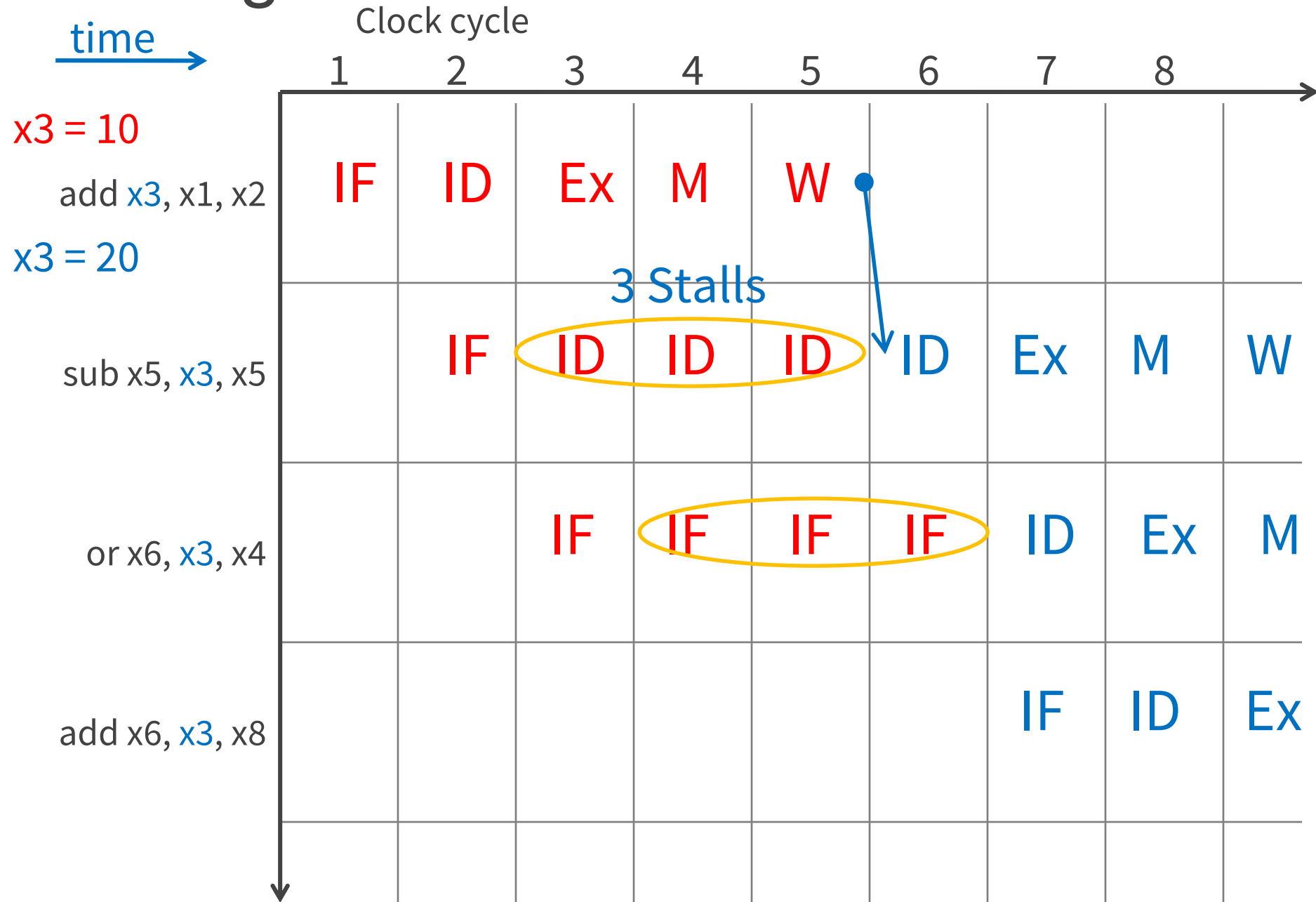
# Stalling



# Stalling



# Stalling

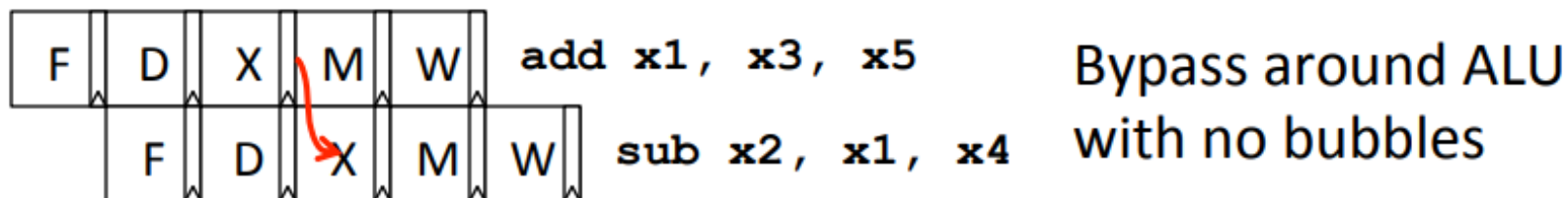
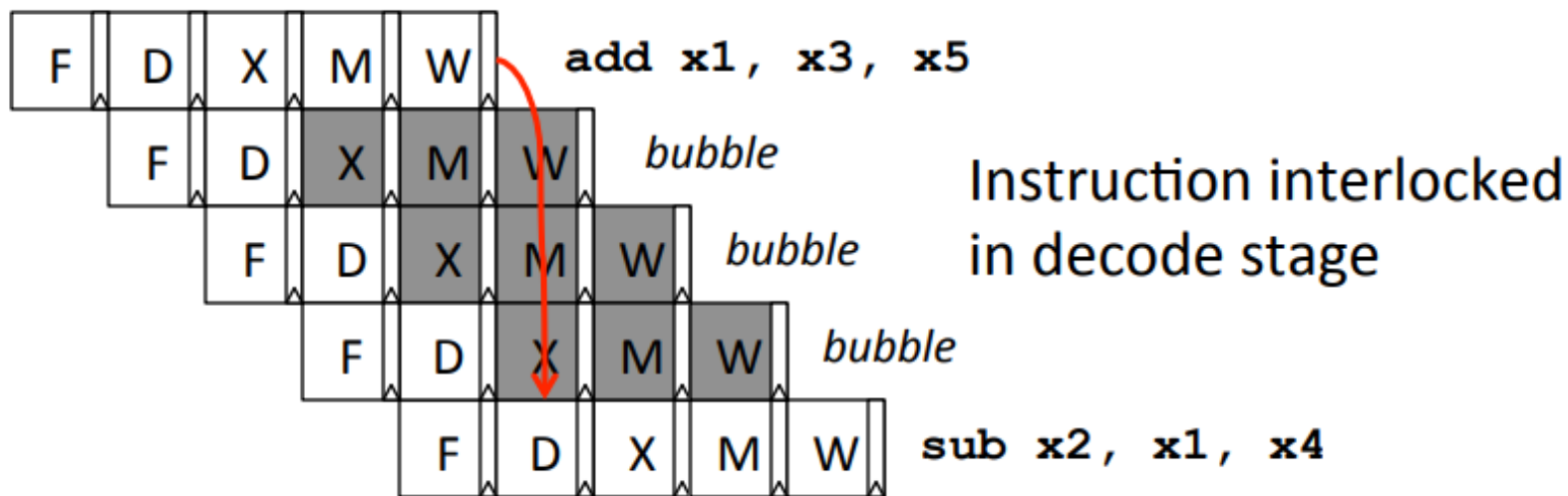


# Stalls and Performance

- CPI increases on every hazard
- Stalls reduce performance
  - But stalls may be required to get correct results
- Are there better solutions?
  - Forwarding/Bypassing

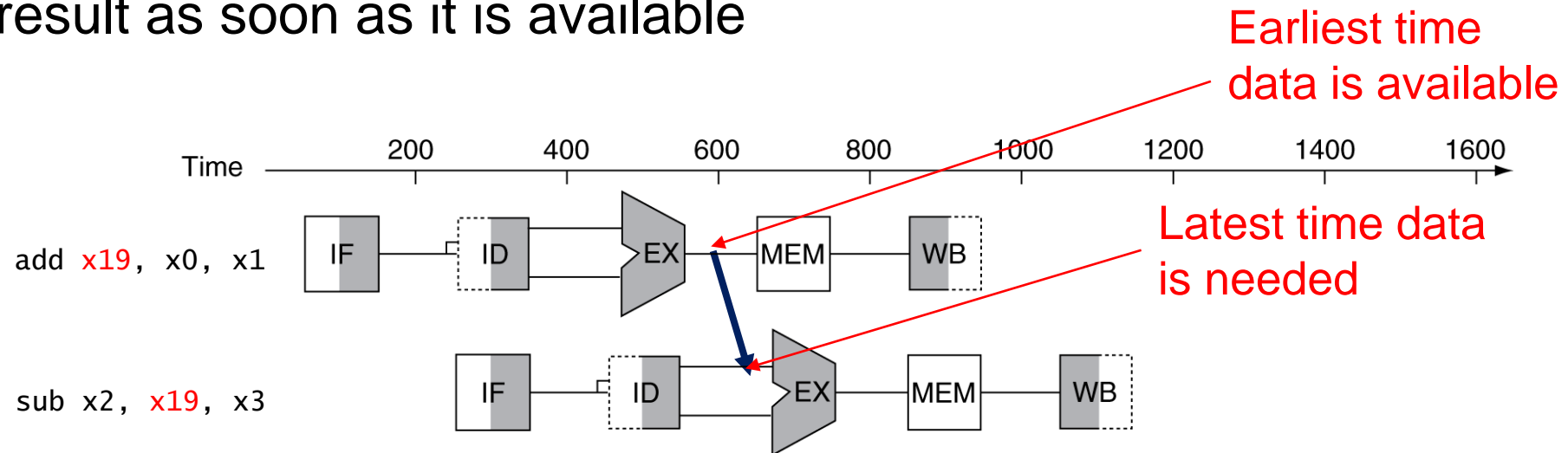
# Interlock/Stall Versus Bypassing

`add x1, x3, x5`  
`sub x2, x1, x4`



# Detect & Forward

- Detection
  - Same as detect and stall, but...
    - each possible hazard requires different forwarding paths
- Forwarding bypasses some pipelined stages forwarding a result to a dependent instruction operand (register).
- Use result as soon as it is available



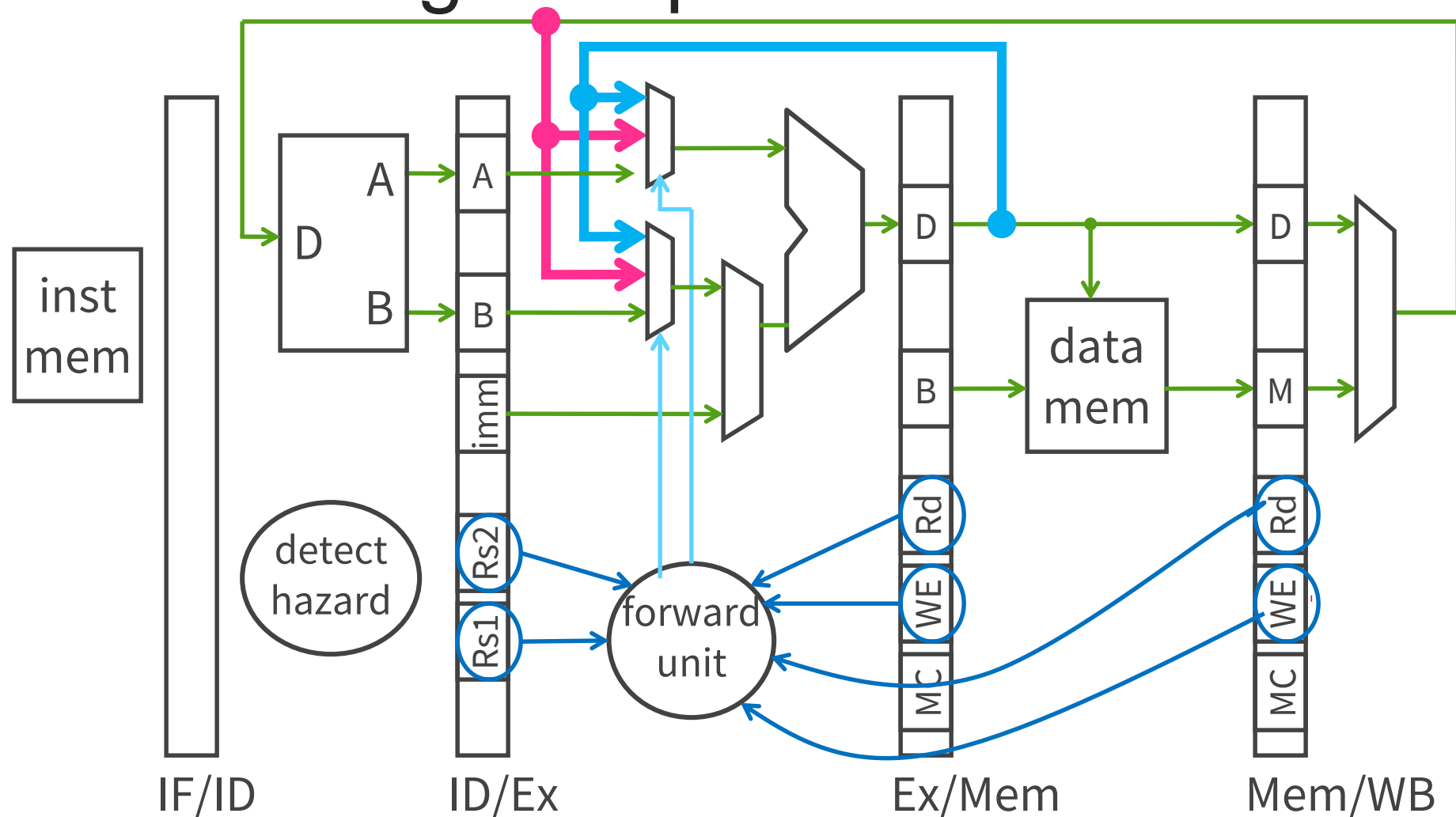
Can't connect input and output of a component directly

# Detect & Forward

- Add mux in front of ALU to select source
- “bypassing logic” often a critical path in wide-issue machines
  - # paths grows quadratically with machine width



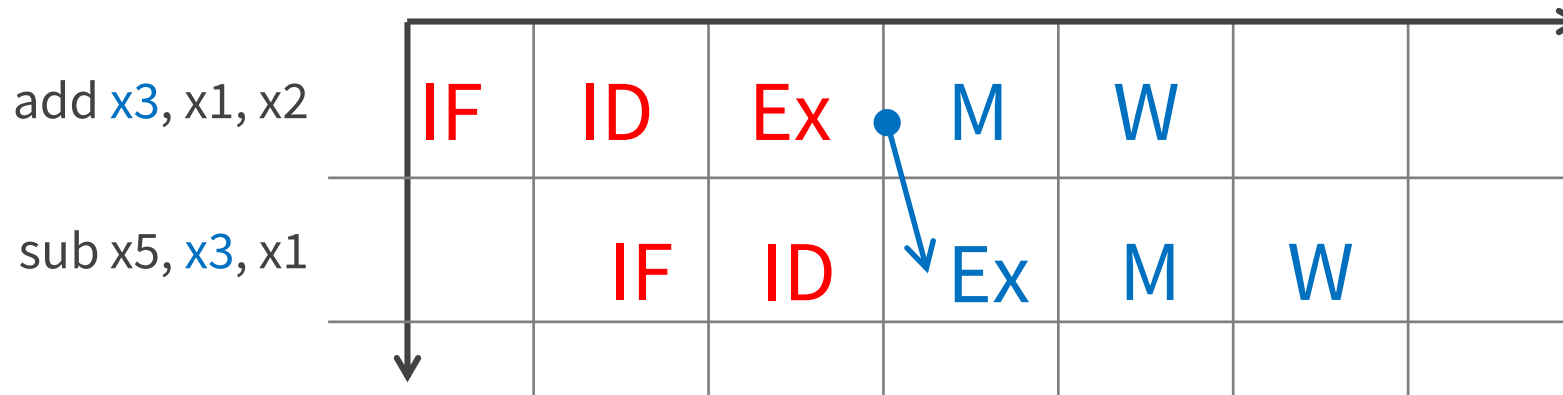
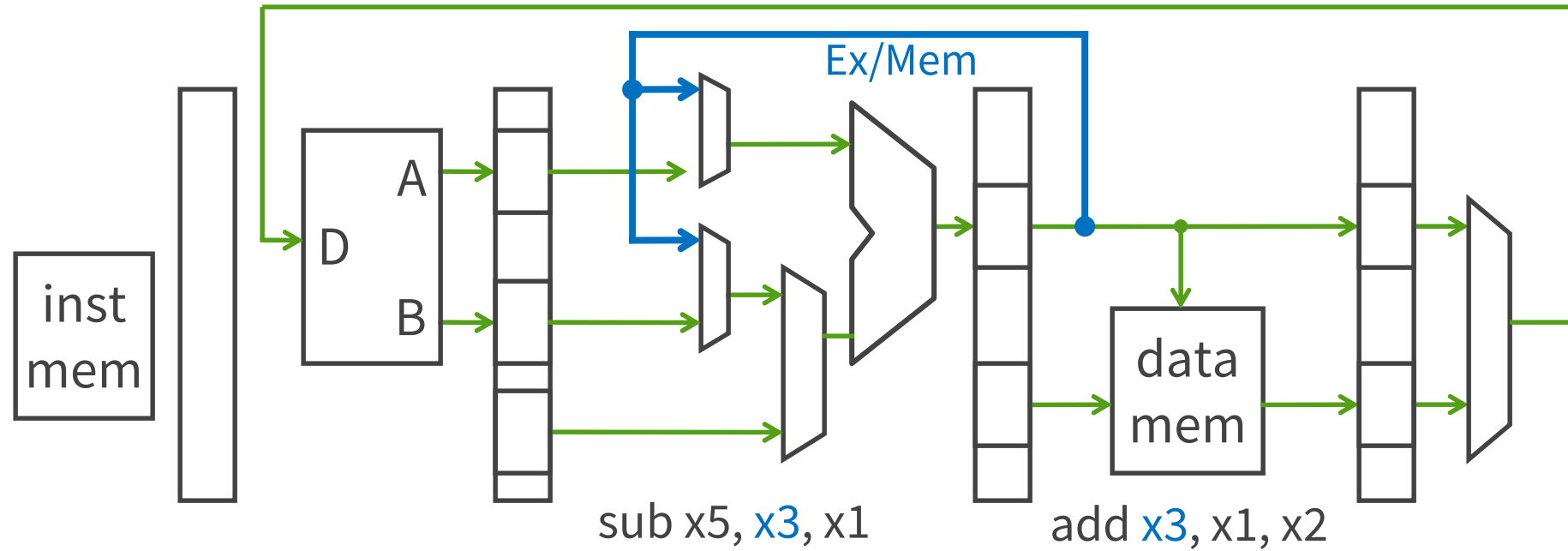
# Forwarding Datapath



Three types of forwarding/bypass

- Forwarding from Ex/Mem registers to Ex stage (M → Ex)
- Forwarding from Mem/WB register to Ex stage (W → Ex)
- RegisterFile Bypass

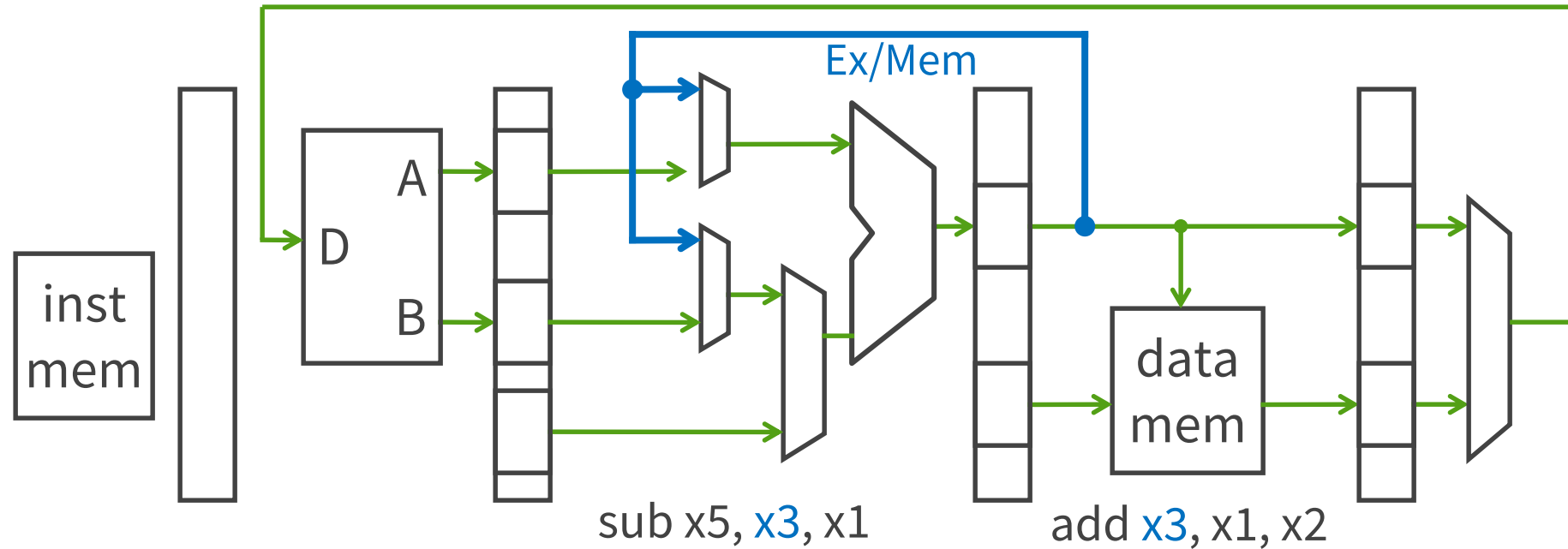
# Forwarding Datapath 1: Ex/MEM → EX



Problem: EX needs ALU result that is in MEM stage

Solution: add a bypass from EX/MEM.D to start of EX

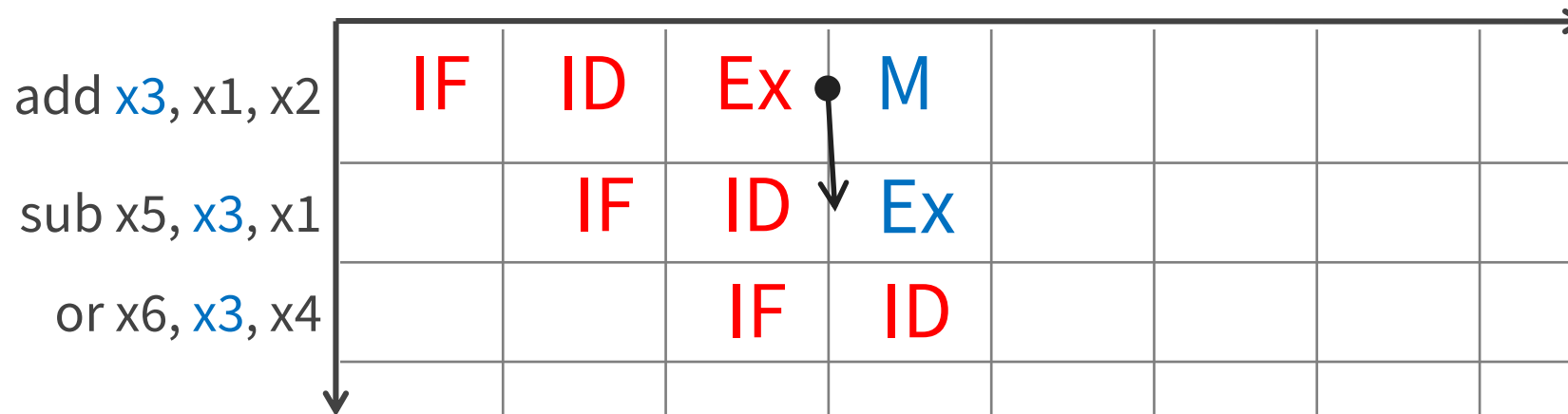
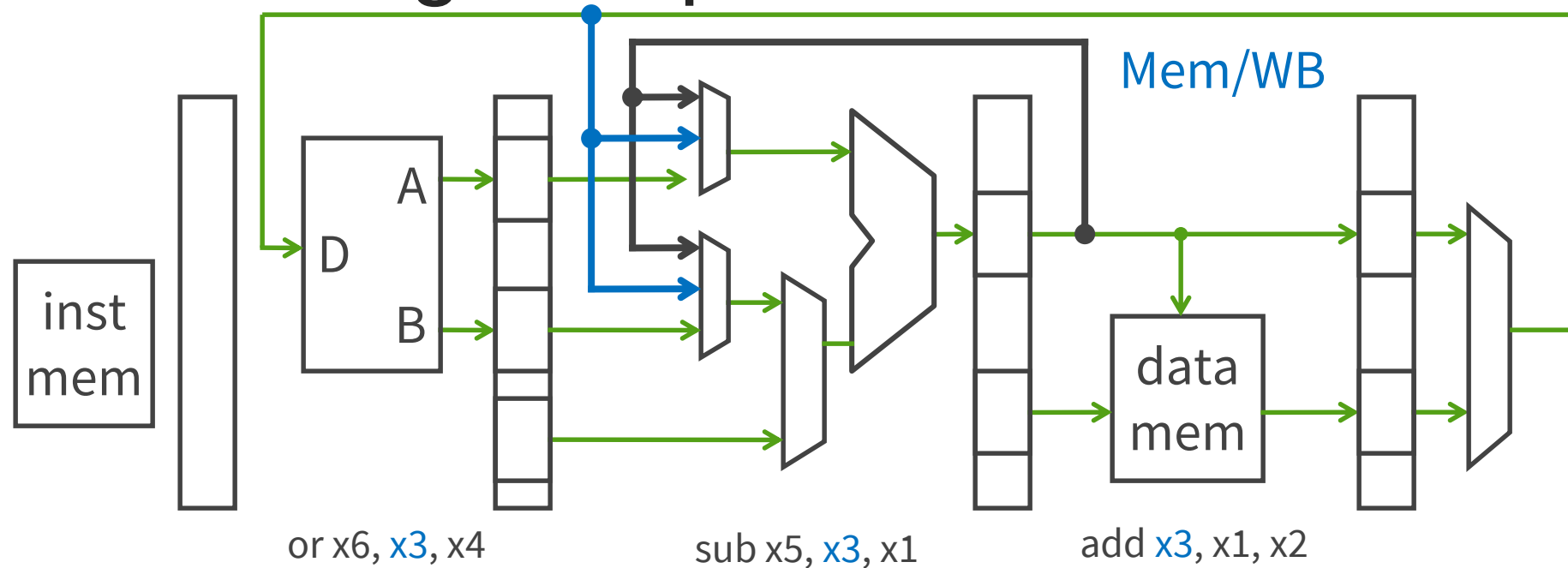
# Forwarding Datapath 1: Ex/MEM → EX



## Detection Logic in Ex Stage:

forward = (Ex/M.WE && EX/M.Rd != 0 &&  
ID/Ex.Rs1 == Ex/M.Rd)  
|| (same for Rs2)

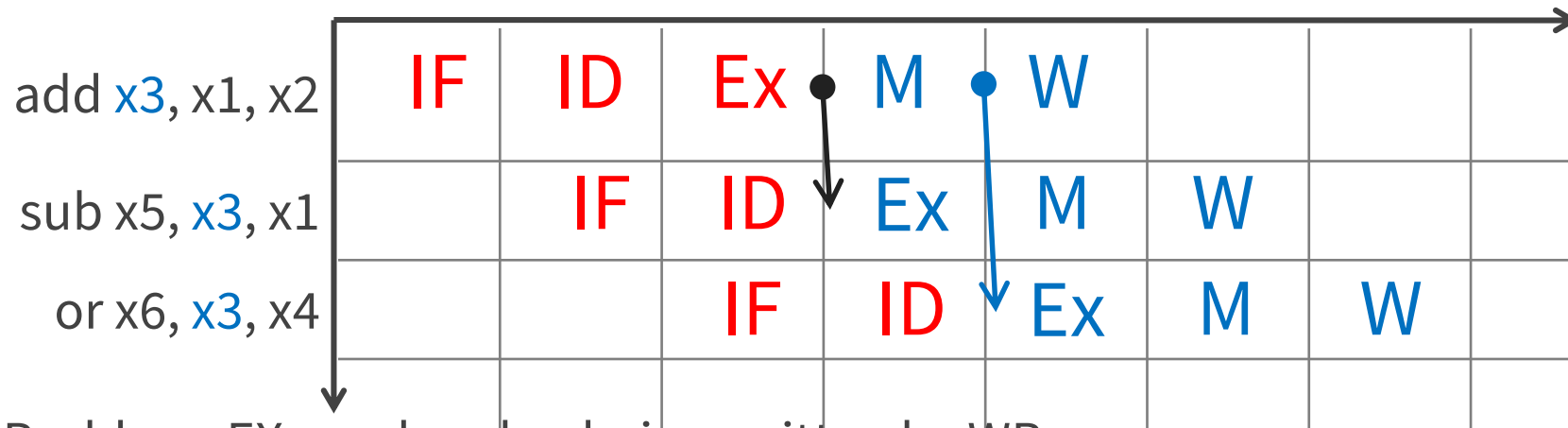
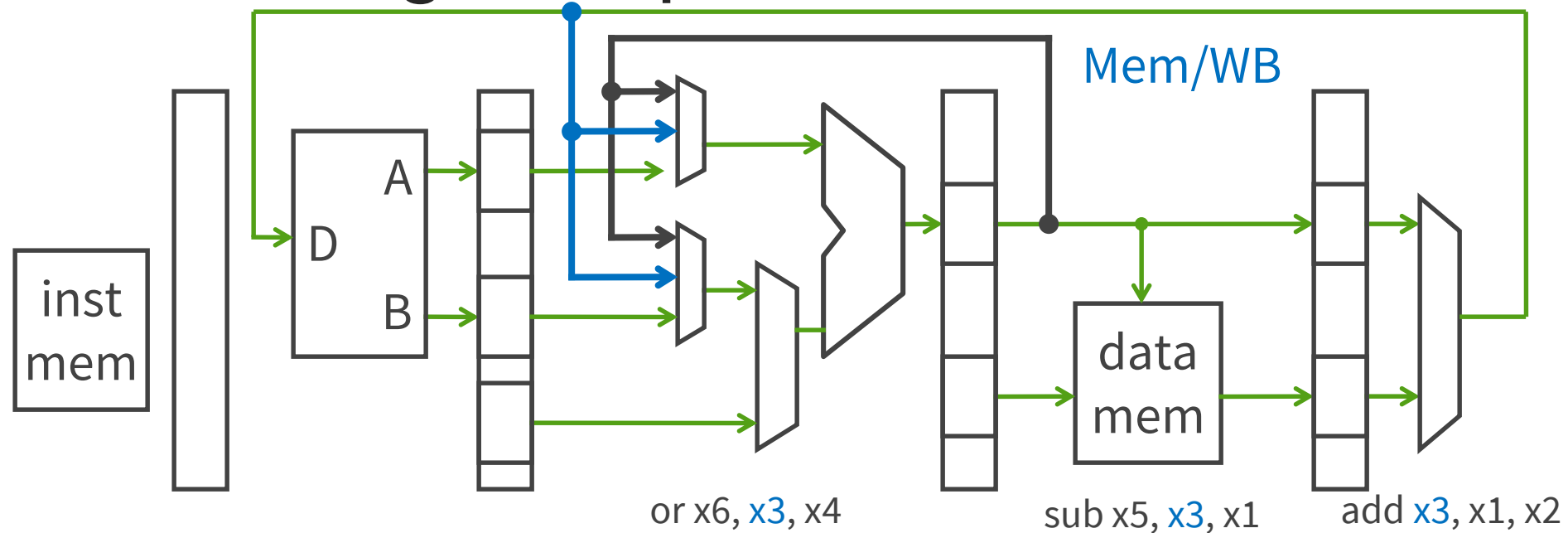
# Forwarding Datapath 2: Mem/WB → EX



Problem: EX needs value being written by WB

Solution: Add bypass from WB final value to start of EX

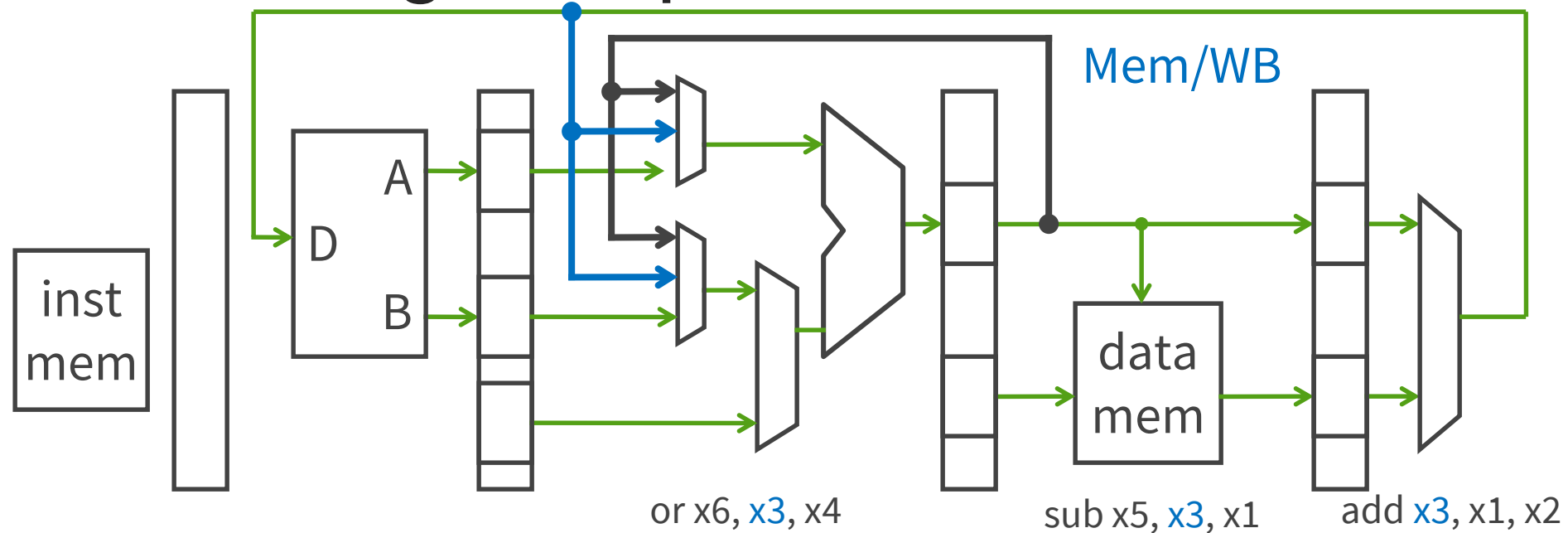
# Forwarding Datapath 2: Mem/WB → EX



Problem: EX needs value being written by WB

Solution: Add bypass from WB final value to start of EX

# Forwarding Datapath 2: Mem/WB → EX



## Detection Logic:

forward = (M/WB.WE && M/WB.Rd != 0 &&

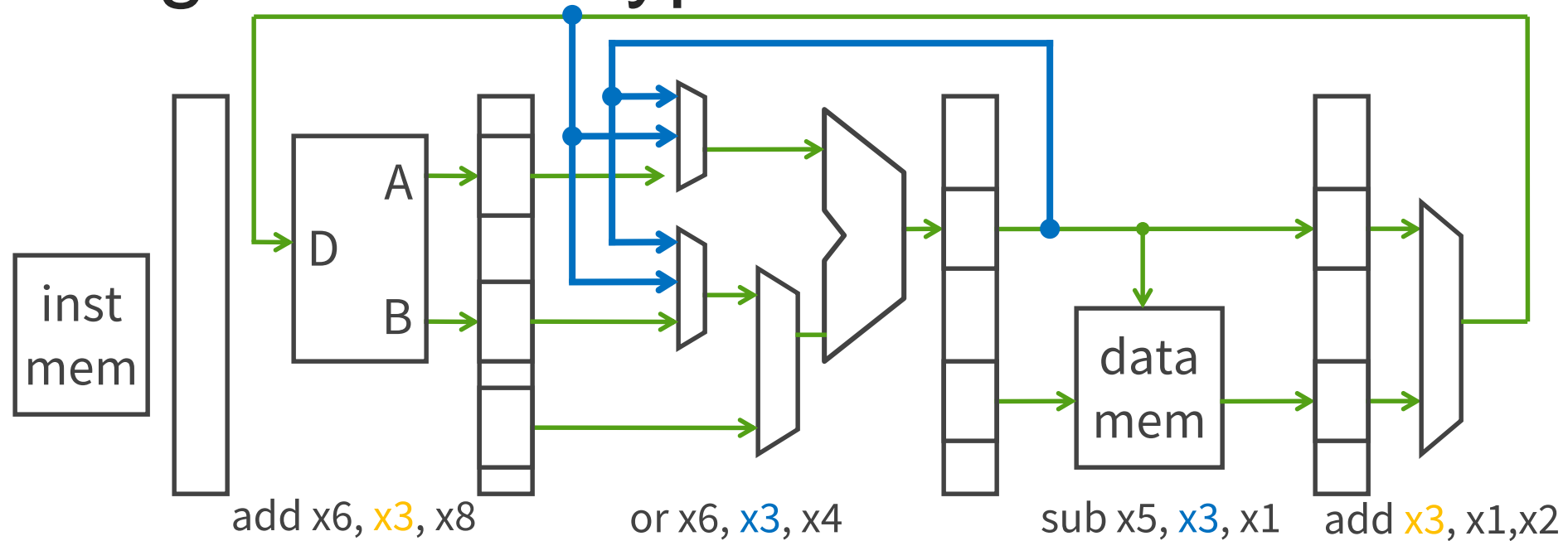
ID/Ex.Rs1 == M/WB.Rd &&

not (Ex/M.WE && Ex/M.Rd != 0 &&

ID/Ex.Rs1 == Ex/M.Rd)

|| (same for Rs2)

# Register File Bypass

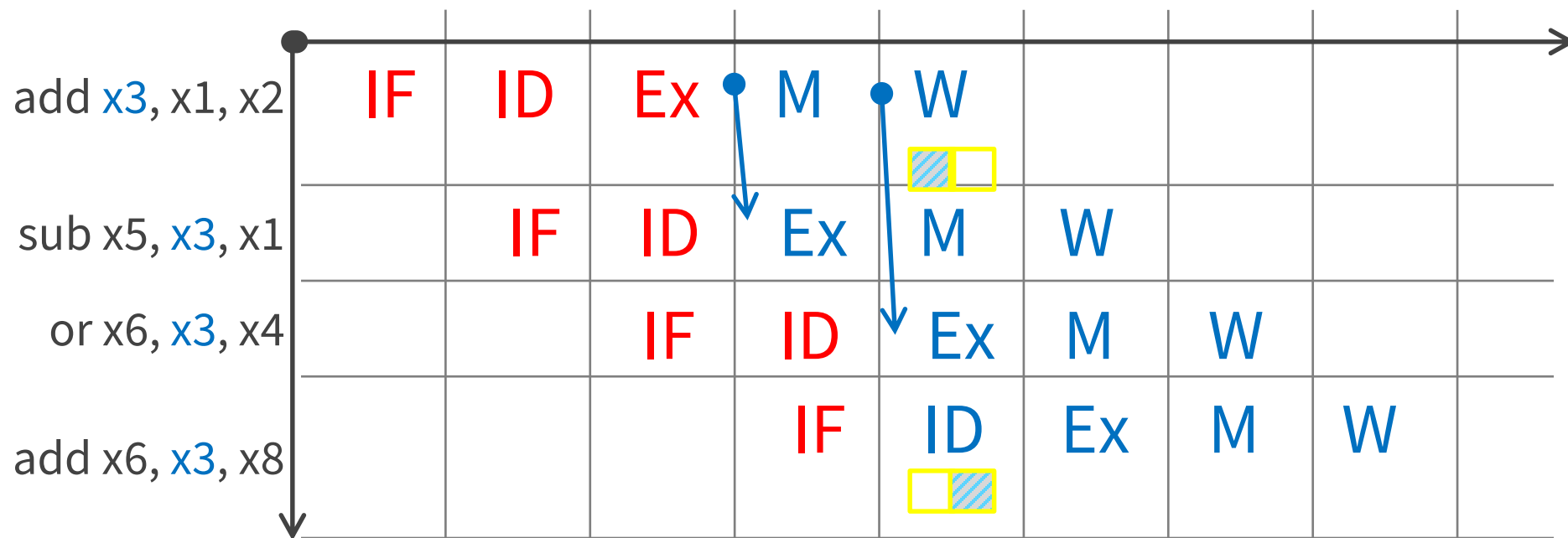
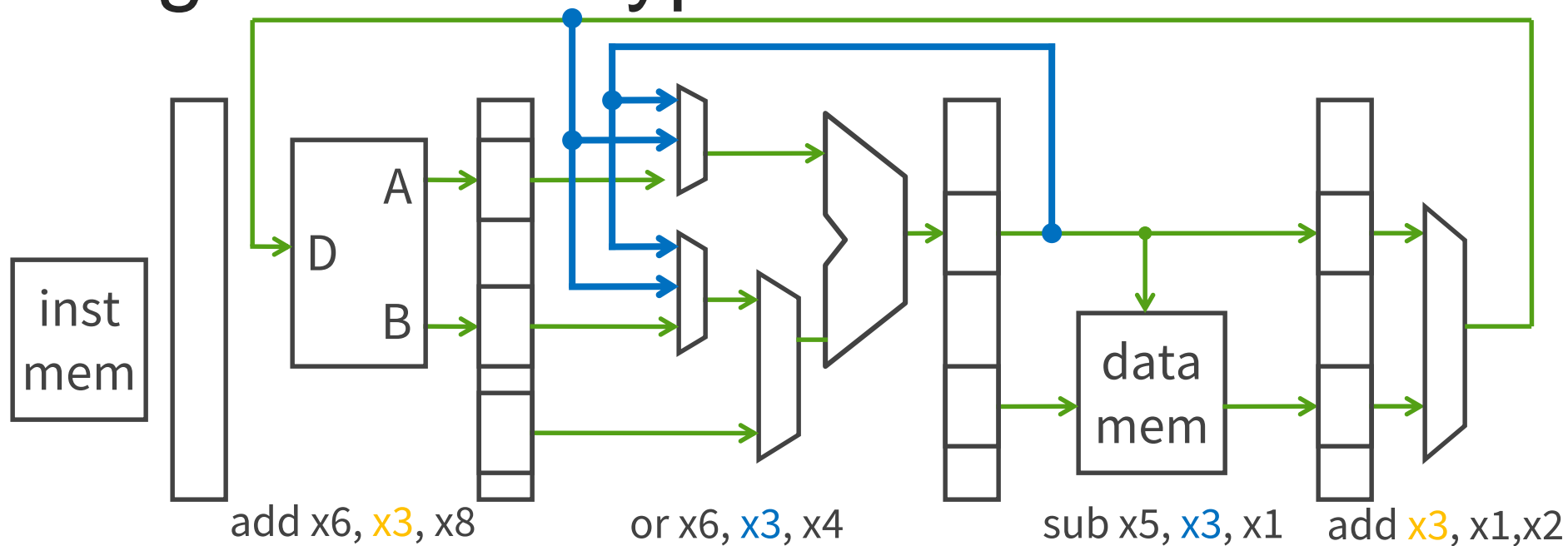


Problem: Reading a value that is currently being written

Solution: **just negate register file clock**

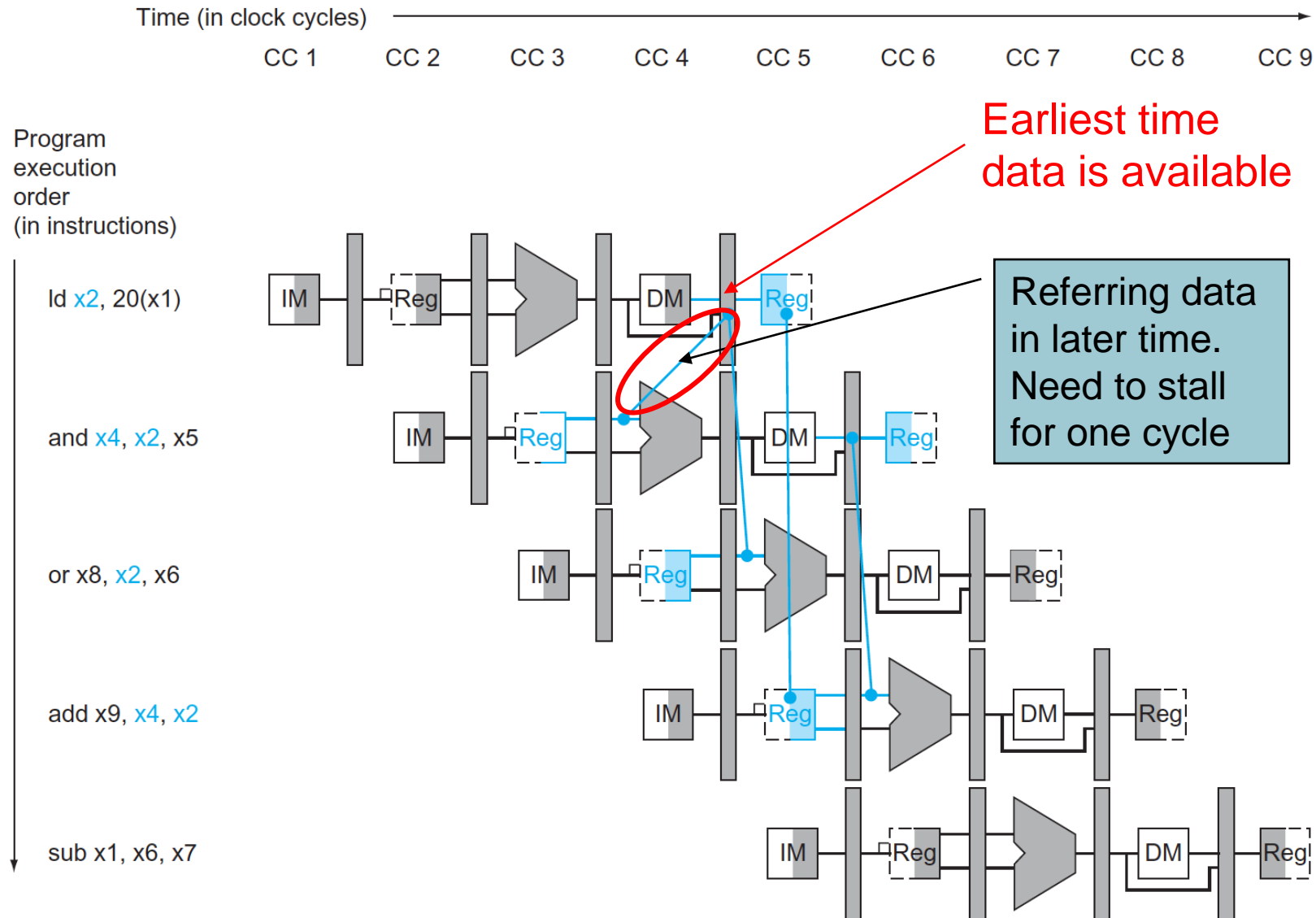
- writes happen at end of first half of each clock cycle
- reads happen during second half of each clock cycle

# Register File Bypass





# Forwarding May Fail



# Load-Use Hazard Detection

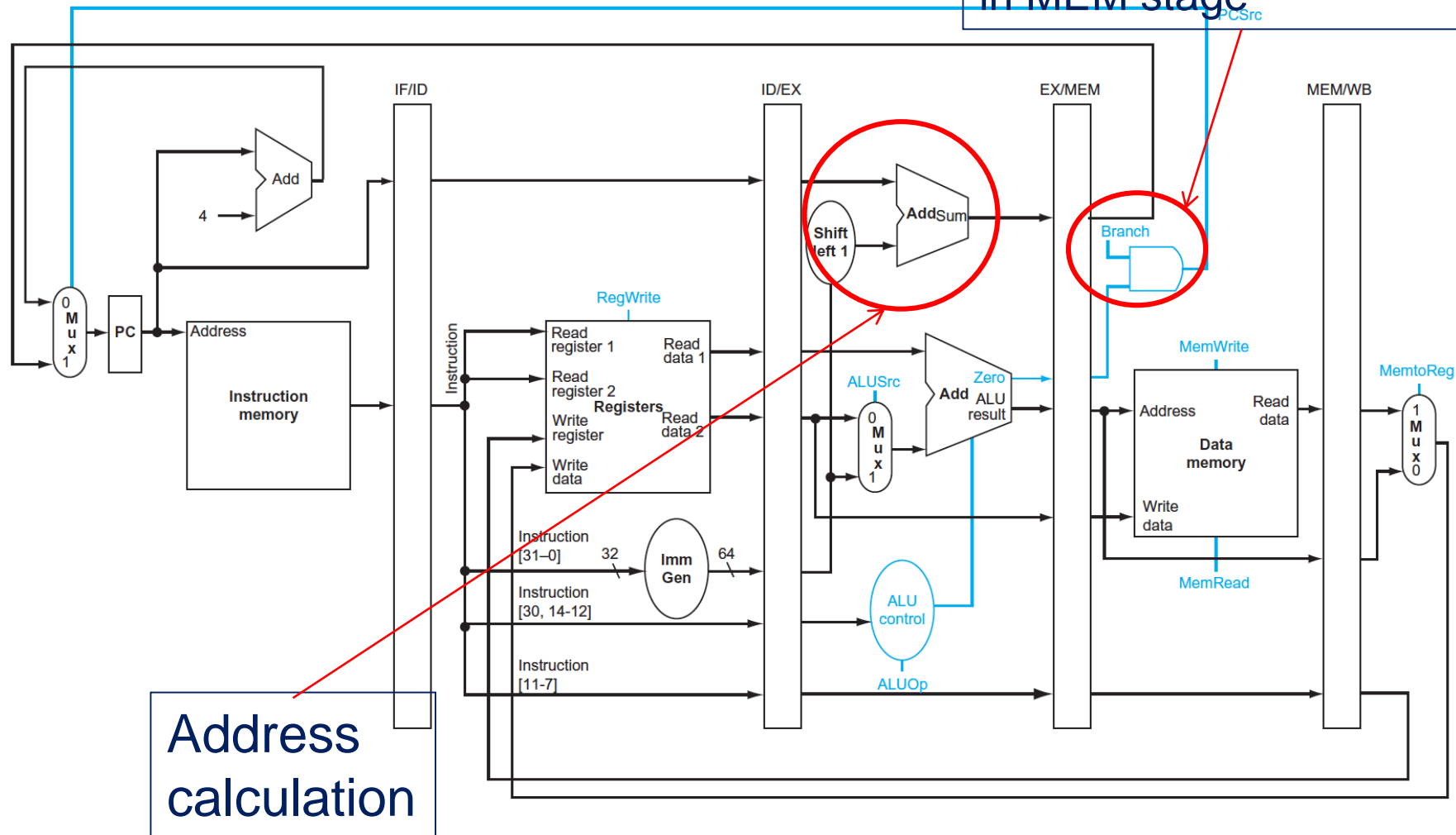
- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
  - ID/EX.MemRead and  
((ID/EX.RegisterRd = IF/ID.RegisterRs1) or  
(ID/EX.RegisterRd = IF/ID.RegisterRs2))
- Hazard Detection is in **ID** stage
- **If detected, stall and insert bubble -> Load delay slot**

# Data Hazard Summary

- Stall
  - Pause current and all subsequent instructions
- Forward/Bypass
  - Try to steal correct value from elsewhere in pipeline
  - Otherwise, fall back to stalling or require a delay slot
- Detection is important!

# Revisit Pipelining

## Determination for branch in MEM stage



# Control Hazards

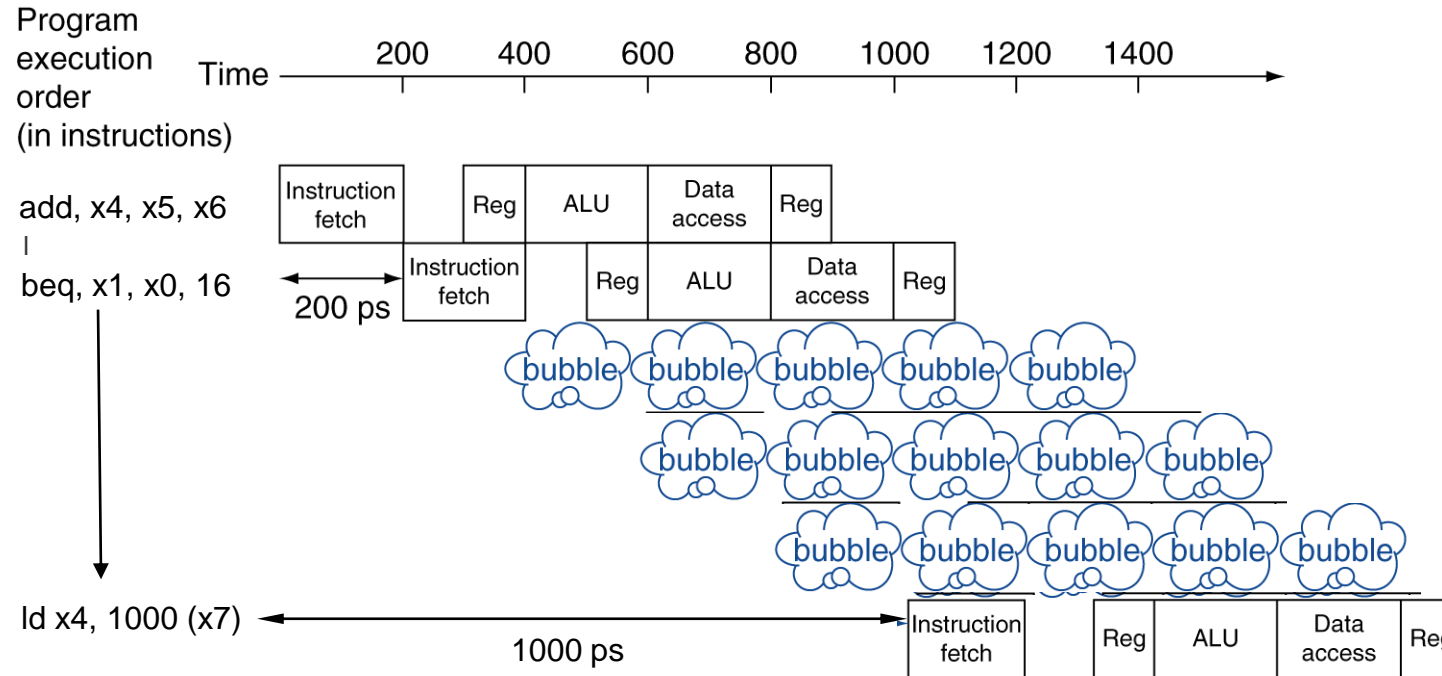
- RISC-V Branch Instruction
  - `beq x10, x11, 2000` // if `x10 == x11`, go to location with respect to `2000ten = 0111 1101 0000`
- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction (Branch Hazard) because
    - When Branch instruction is still in the **ID** stage, target instruction is needed in the **IF** stage

# Handling Control Hazards

- Stall on branch
- Always assume branch not taken or taken
- Branch prediction
- Delayed Branch

# Stall on Branch

- Wait until branch outcome determined before fetching next instruction



# Assume Branch Not Taken

- If we are right, lucky us!
- If we are wrong, penalty will be to flush some (up to 3) instructions
- Penalty may be reduced
  - By making earlier decision on branch or not
  - Need to compare registers and compute target early in the pipeline



# Branch Prediction

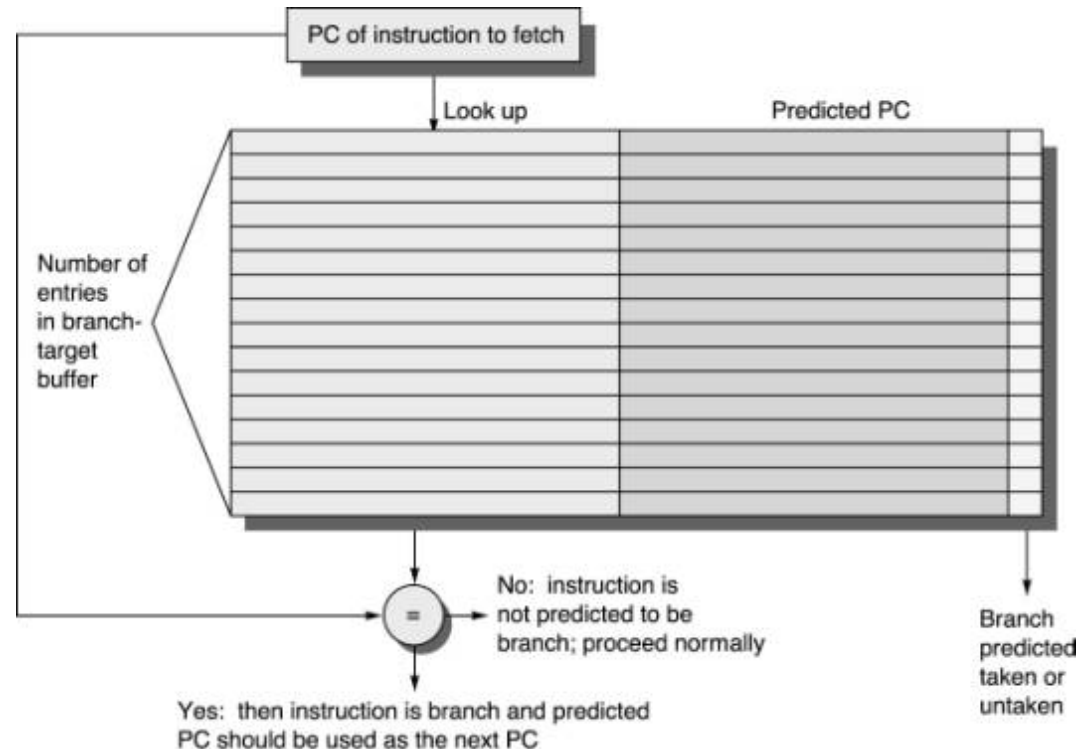
- Static prediction
  - Based on typical branch behavior
  - All decisions are made at compile time
  - Example: loop and if-statement branches
    - Could predict backward branches taken
    - Could predict forward branches not taken
- Dynamic prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - If wrong, take penalty, and update history

# 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

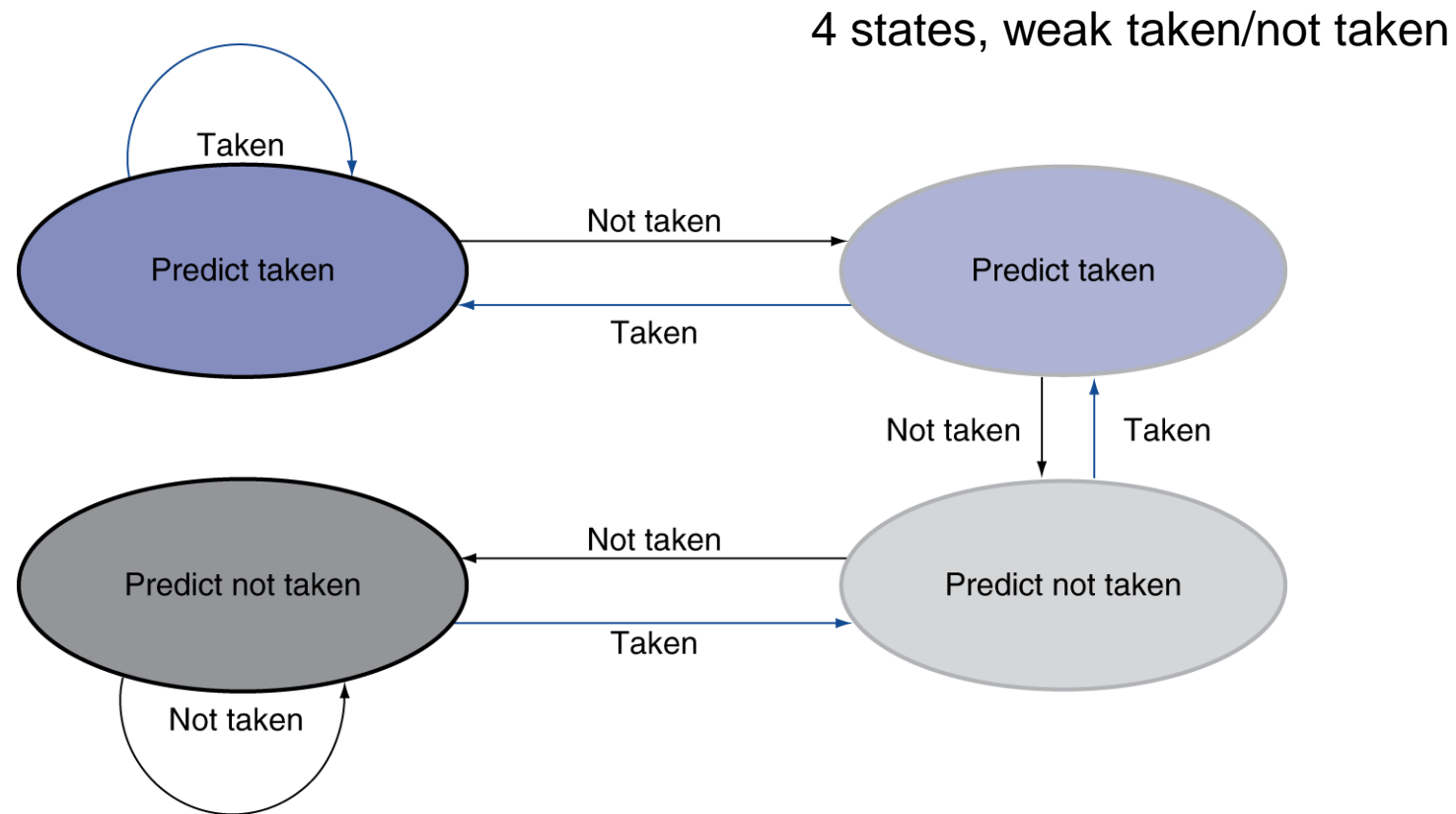
# Dynamic Branch Prediction Components

- Branch prediction/target buffer or branch history table (BTB)
  - A small memory indexed by the lower portion of the address of the branch instruction
  - The memory contains a bit that says whether the branch was recently taken or not. (If the prediction is incorrect, the prediction bit is inverted and stored back. )



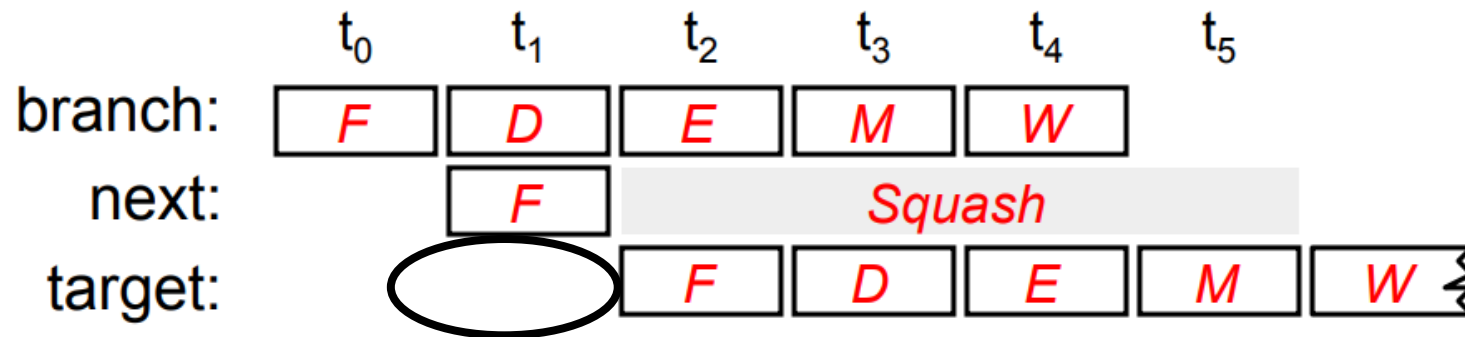
# 2-Bit Predictor

- Only change prediction on two successive mispredictions



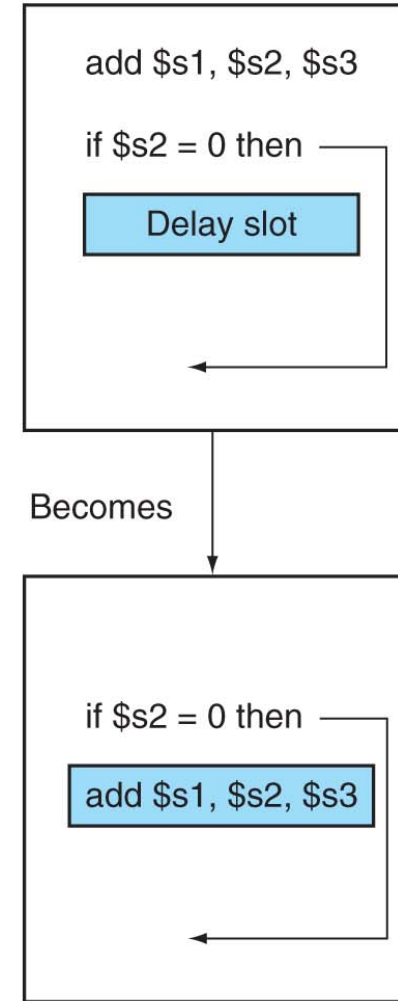
# Delayed Branch

- Always delay the branch
  - With one instruction, NOP or usually a real instruction, for 5 stage pipeline
  - Maybe more delays for deeper pipeline
- Requires carefully designed compiler



# Example for Delayed Branch

- Will remove the 1 clock cycle penalty
- Will work only if instructions can be found to fill the delay slot



# Delayed Branch

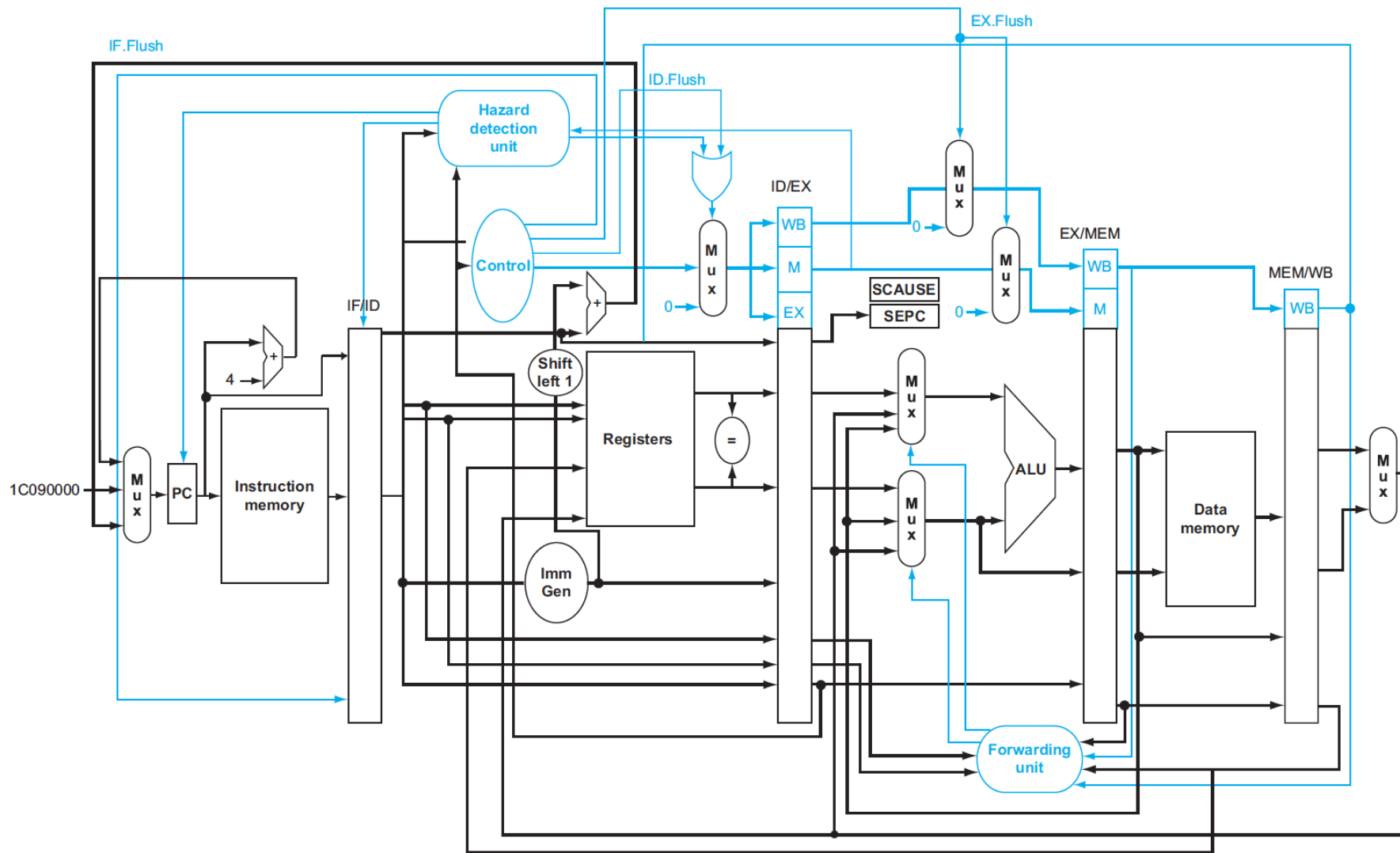
- Always execute the instruction immediately following branch
  - Called Branch delay slot
  - Further reduce branch penalty
- One branch delay slot becomes insufficient
  - When pipeline gets deeper or multiple instructions per clock cycle
  - Dynamic branch prediction is better choice

# Pipeline Hazard Checklist (So far)

- Data dependencies (memory/register)
  - True Data Dependence (RAW)
  - Anti Dependence (WAR)
  - Output Dependence (WAW)
- Control Dependences



# Where we are?



Five-stage in-order pipelined processor (End of VE370)...

# Where are we Heading?

- T4: Advanced Processors I

# Acknowledgement

Slides in this topic are inspired in part by material developed and copyright by:

- ARM Courseware
- Prof. Ron Dreslinski @ UMich, EECS 470
- Prof. Hakim Weatherspoon @ Cornell, CS 3410
- Prof. Krste Asanovic @ UCB, CS252
- Xinfei Guo @ JI, VE370 2021 SU

# Action Items

---

- HW#1 is upcoming
- Review pipeline & hazards topics from VE370
- Reading Materials
  - Ch. 3.1, 3.2
  - P&H, Computer Organization and Design RISC-V Edition, Ch. 4.5 – 4.6, 5.7