CENG 3420
Computer Organization & Design
Lecture 10: Hazard

Textbook: Chapter 4.8-4.10

Zhengrong Wang

CSE Department, CUHK

zhengrongwang@cuhk.edu.hk

Data Hazards

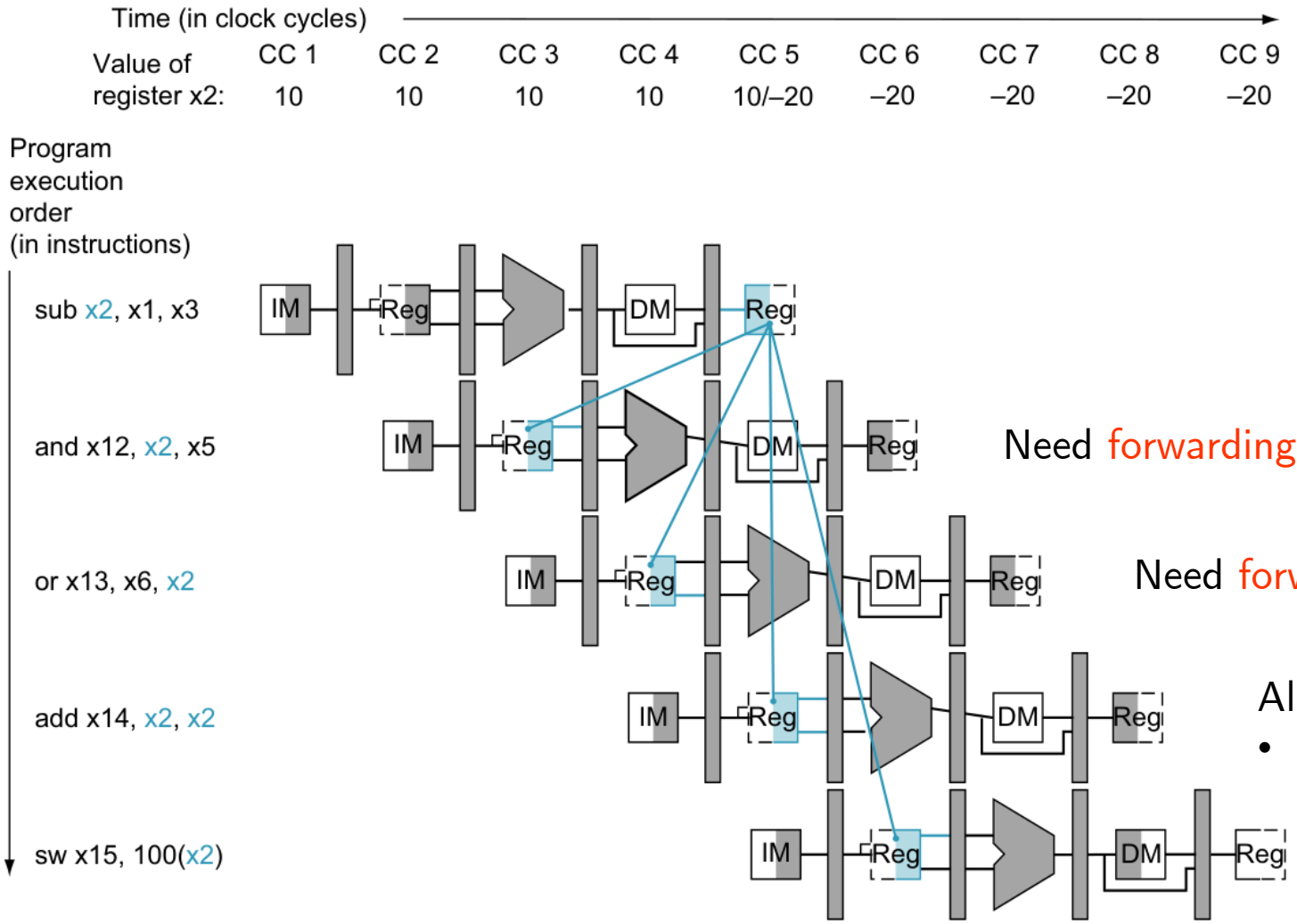
Data Hazards

- Happens when instruction uses result from prior instructions.

```
sub     x2, x1, x3     // Register x2 written by sub
and     x12, x2, x5    // Rs1 depends on sub
or      x13, x6, x2    // Rs2 depends on sub
add     x14, x2, x2    // Rs1 and Rs2 depends on sub
sw      x15, 100(x2)   // Base depends on sub
```

Data Hazards

- Happens when instruction uses result from prior instructions.



Need forwarding from EX/MEM stage registers.

Need forwarding from MEM/WB stage registers.

Already handled by register file.
- Write and read in the same cycle.

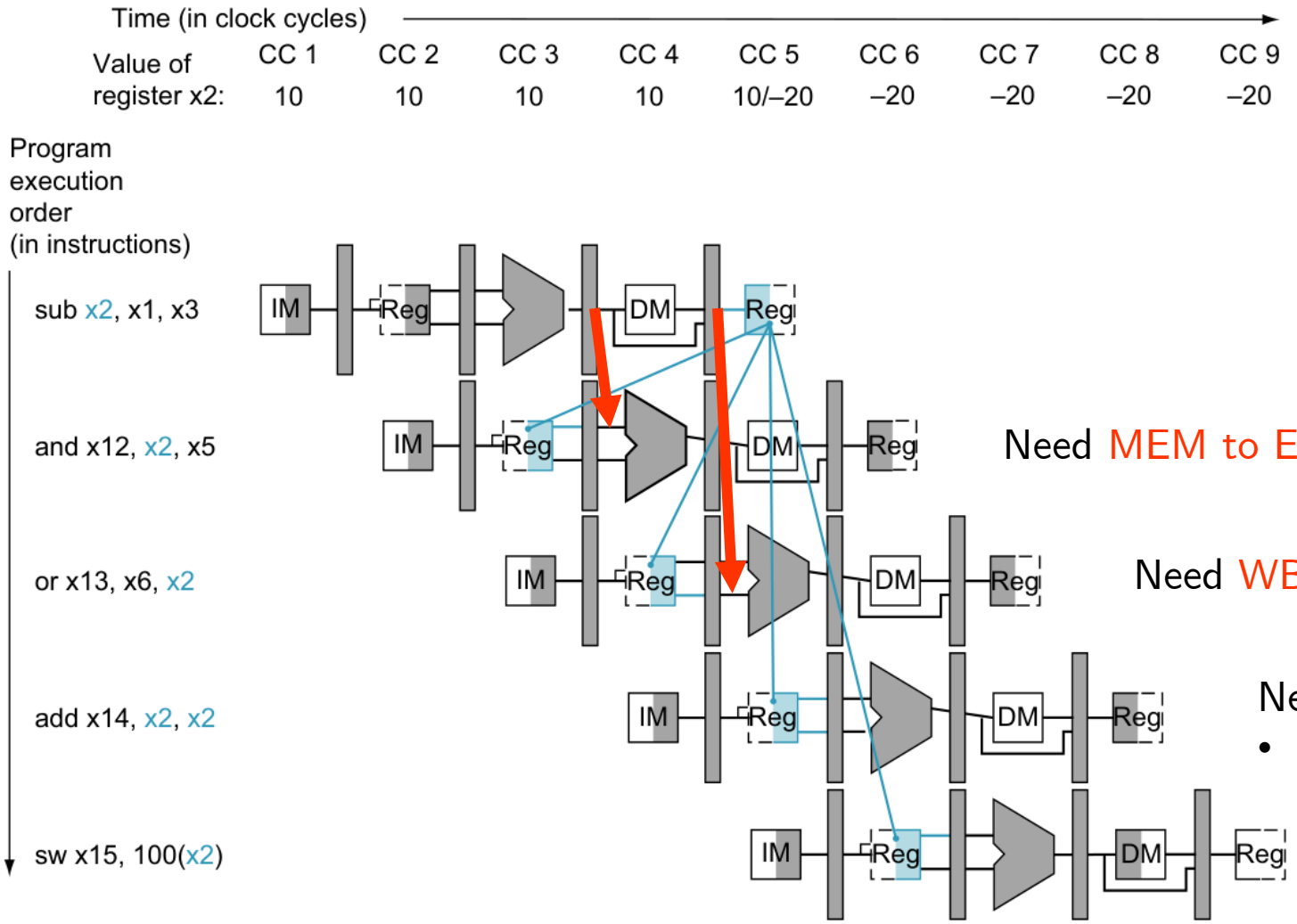
No hazard

We Need Data Forwarding

- Terminology: we name the pipeline state register field as "A/B.Field":
  - A/B means the state register between stage A and stage B.
  - E.g., ID/EX.RegisterRs1 → register id of rs1 in ID/EX pipeline register.
- No need to forward: When Prior instruction's rd ≠ next instruction's rs1/rs2
  - No data dependence at all.
- Otherwise, two types of forwarding
  - Forward from MEM stage to EX stage
  - Forward from WB stage to EX stage
  - Why forward to EX? EX is the **latest** point that must have the correct value!
  - Note: WB to ID hazard is automatically handled by the register file
- What are the forwarding type we need in prior example?

Data Hazards

- Happens when instruction uses result from prior instructions.



Need MEM to EX forwarding.

Need WB to EX forwarding.

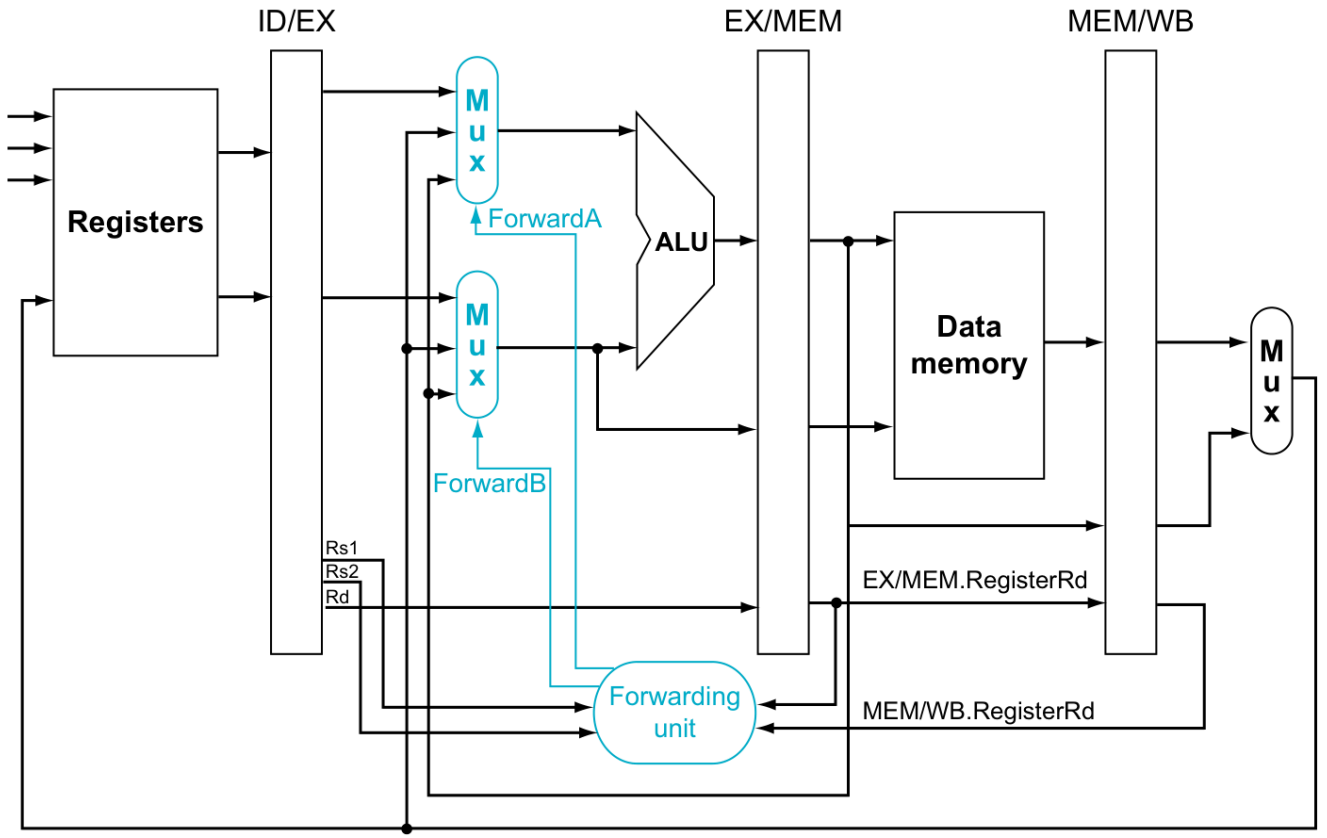
Need WB to ID forwarding.

- Write and read in the same cycle.

No hazard

Datapath with Forwarding

- Forward = 00 → No forwarding, operand from register file.
- Forward = 10 → MEM to EX forwarding, operand from prior ALU result.
- Forward = 01 → WB to EX forwarding, operand from data memory or earlier ALU result.



Data Forward Condition

- Essentially, we want to check prior **rd** against current **rs1/rs2**.
- MEM to EX condition:
  - We check RegWrite as not every instruction writes to register file (e.g., branch, store).
  - We check it's not writing to x0, since x0 is always 0 and no forwarding required.

```
If  (EX/MEM.RegWrite
and (EX/MEM.RegsiterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRs1)) FowardA = 10
```

- WB to EX condition:

```
If  (MEM/WB.RegWrite
and (MEM/WB.RegsiterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRs1)) FowardA = 01
```

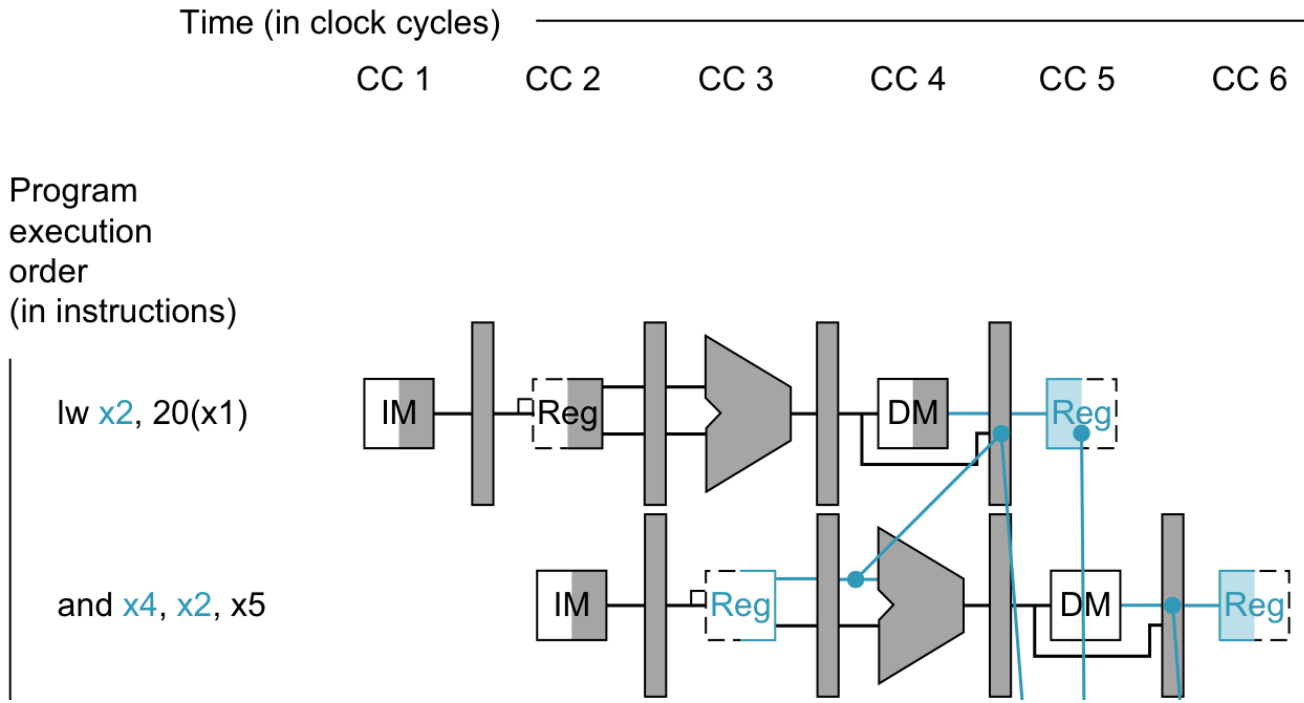
Prioritize MEM to EX Forward

- When both MEM and WB needs to forward to EX, prioritize MEM.
  - MEM is the latest instruction, so it has the most updated result.
- Updated WB to EX condition:
  - Check if no MEM to EX forward happening.

```
If  (MEM/WB.RegWrite
and (MEM/WB.RegsiterRd != 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and (MEM/WB.RegisterRd == ID/EX.RegisterRs1)) FowardA = 01
```

Forward vs Stall

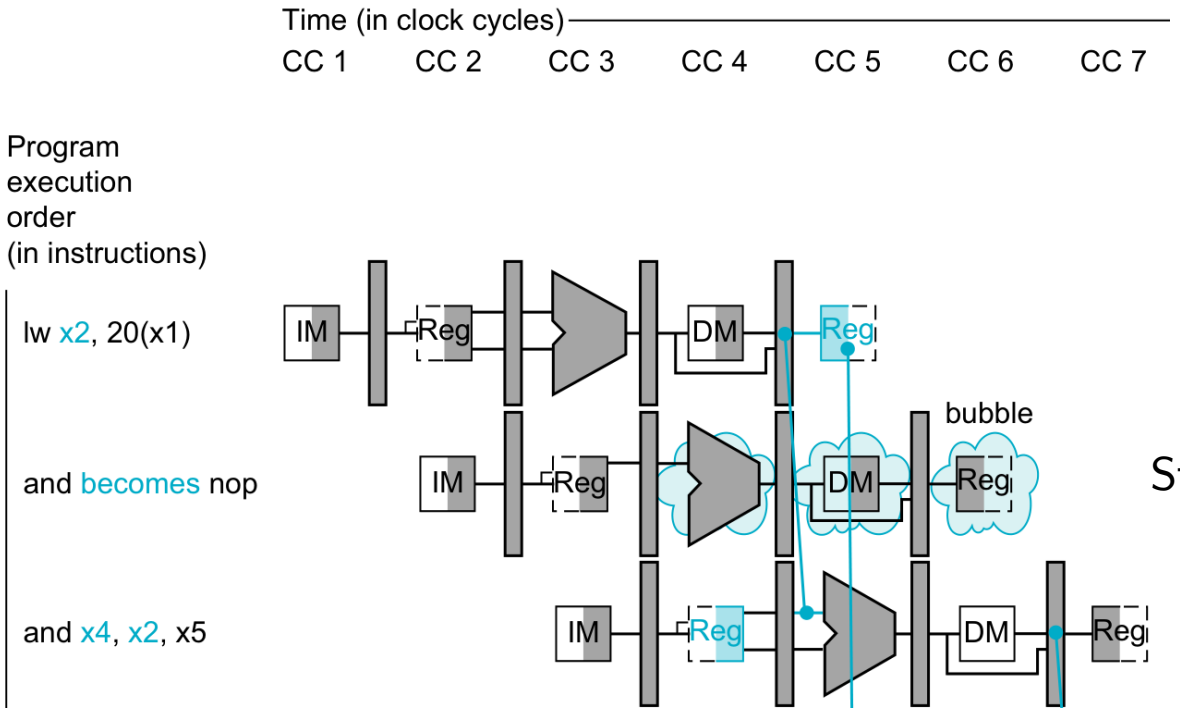
- Forward cannot solve all data hazards!
- Load followed by immediate user instruction → Can not forward.
  - Value is loaded after MEM stage at cycle 4, so only be ready at cycle 5.
  - The next user instruction need the value at EX stage at cycle 4.



Stall Immediate User of Load Instruction

- Must stall the next user of a load instruction at ID stage by 1 cycle.

```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
    (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
  stall the pipeline
```

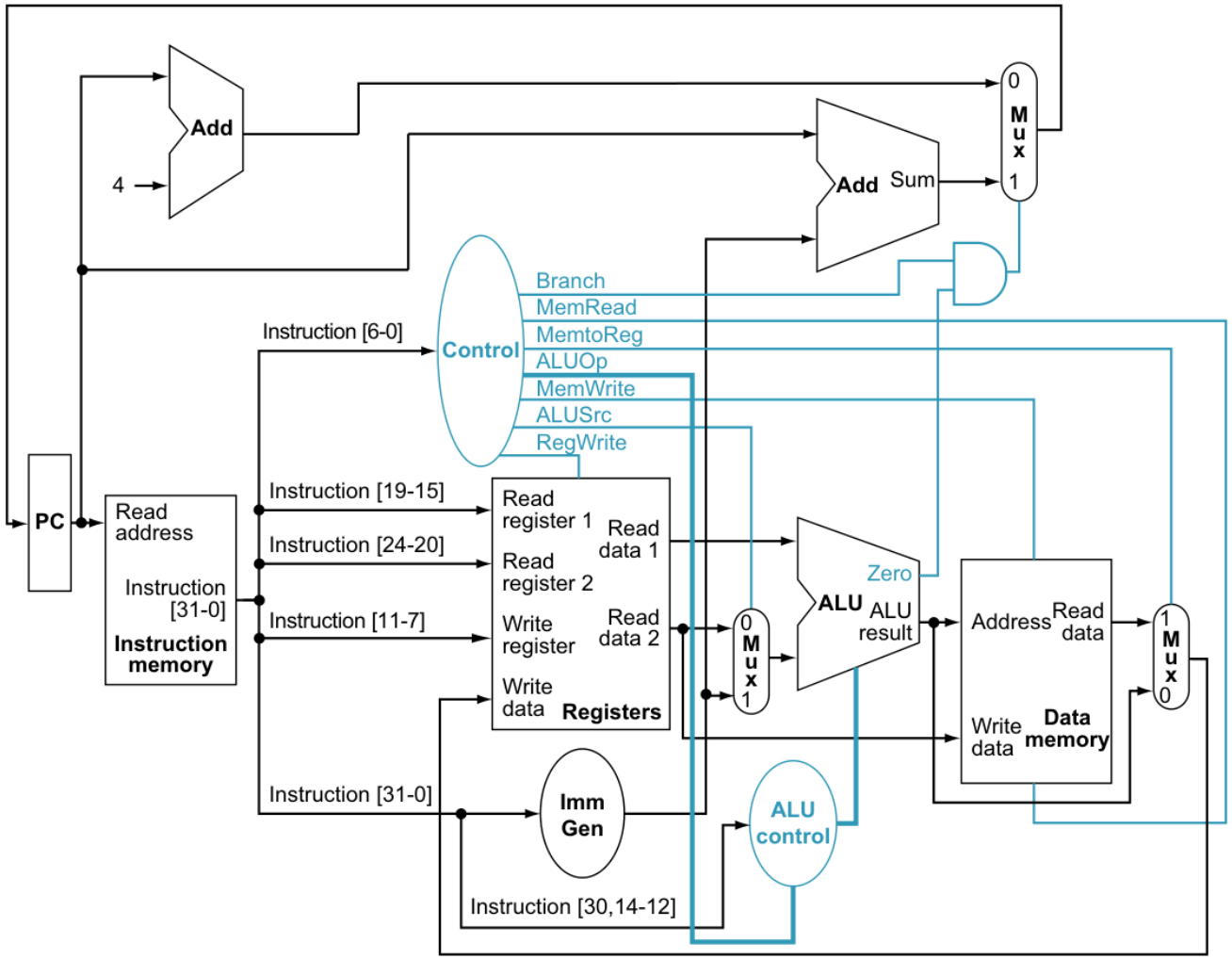


Stall the immediate user at EX stage

Resume the immediate user from EX stage

How to Stall?

- Setting all control signals to zero → Turning this instruction into nop.

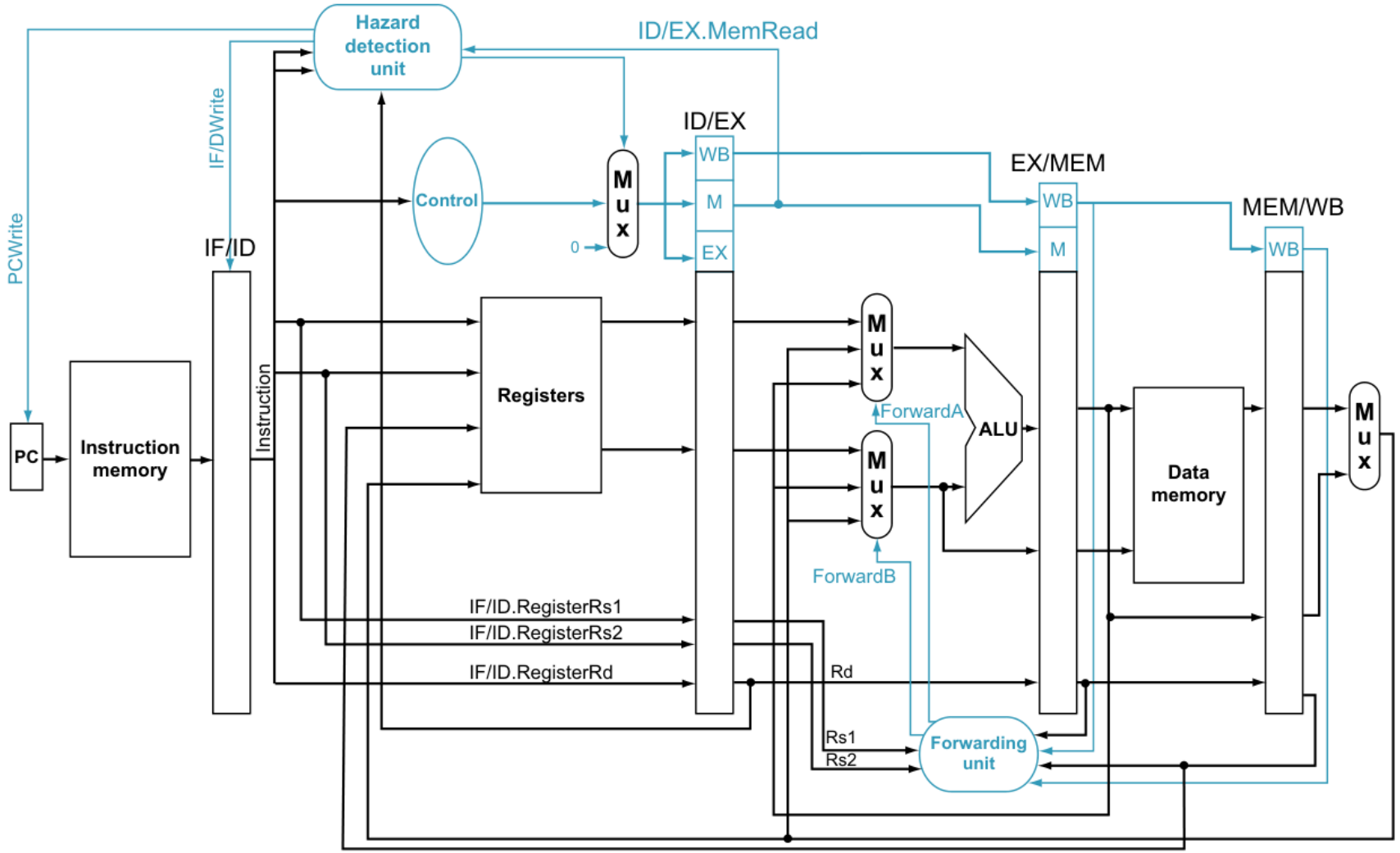



```
RegWrite = 0 → Do not write reg
MemRead = 0  → Do not read mem
MemWrite = 0 → Do not write mem
Branch = 0   → Do not branch

Does nothing → nop instruction.
```

Datapath with Forward and Stall

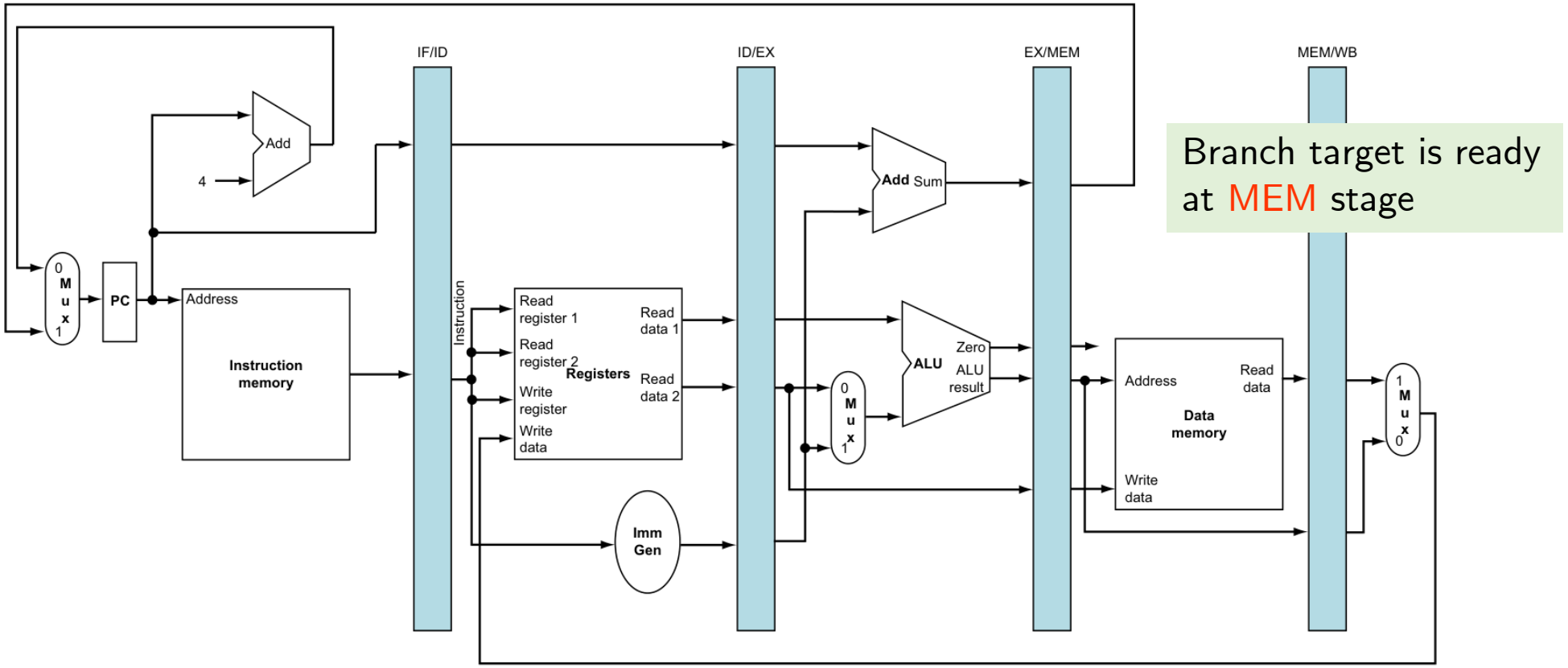
- Add Hazard detection unit to stall the pipeline when needed.



Control Hazard

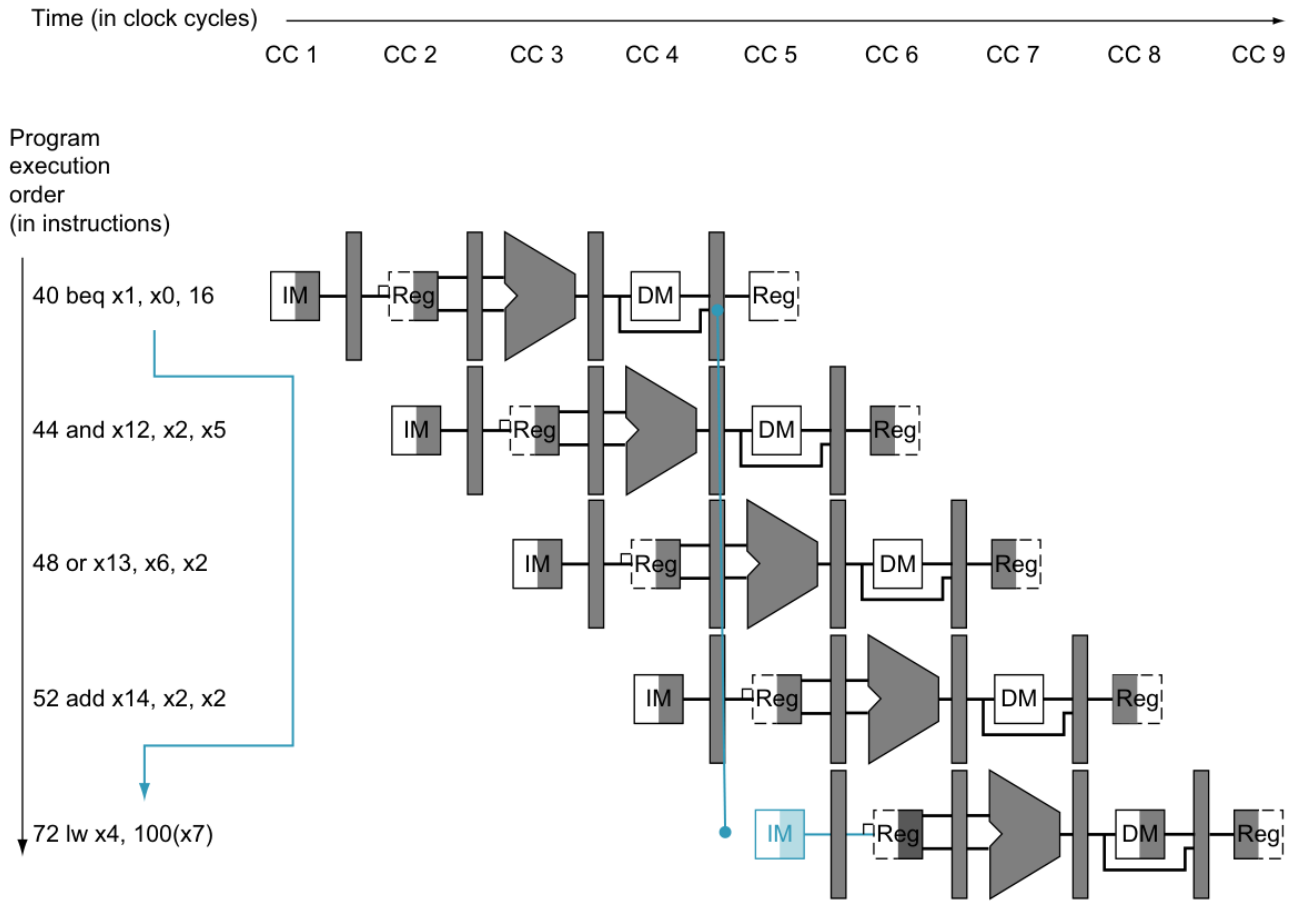
Need to Flush the Pipeline if Branch

- The datapath assumes branch not taken – always fetch PC + 4 sequentially.
- However, if a branch is taken, we must flush the pipeline and restart.
- Since the branch target is resolved at EX stage, flush IF, ID, EX stage.



How to Flush

- To flush, set all control signals to zero (similar to nop in load-use hazard).
- Add IF.Flush signal to IF stage → Set instruction to all zeros.



Resolve Branch at ID Stage

- If branch is resolved at ID stage, we only need to flush IF/ID stage.
- For branch target: PC and Offset is ready at ID stage.
  - Move the adder to ID stage.
- For branch equality test: XOR the output of register file and check if all 0.
  - Add XOR and a tree of OR gates to ID stage.
- Complication: More data hazard to ID stage.
  - EX to ID forward: `add x2, x1, x3; beq x2, x4, 4(x10);`
  - MEM to ID forward: `add x2, x1, x3; other inst; beq x2, x4, 4(x10);`
  - Load to branch must stall 2 cycles: `lw x2, 4(x1); beq x2, x4, 4(x10);`
  - Note: WB to ID forward is automatically handled by register file.
- This helps reduce overhead of taken branch to 2 cycles.

Dynamic Branch Prediction

- "Assume not taken" is a form of static branch prediction.
  - Can also assume always taken but cannot avoid bubble since branch is resolved later.
- Dynamically predict the branch: guess the branch target and fetch from there.
- Example: 1-bit branch predictor.
  - A small branch prediction buffer at IF stage to remember last branch target of the beq.
  - 1-bit as it flips every time the branch outcome changes.
  - For example, for a loop of 10 iterations, it will mispredict the first and last iteration.
  - Correct rate: $(10-2) \div 10 \times 100\% = 80\%$
- Better: 2-bit branch predictor, only flips if sees consecutive 2 taken/not taken.
  - It is more resilient to occasional misprediction, e.g., breaking out of the loop.
  - For example, for a loop of 10 iterations, it will mispredict the last iteration.
  - Correct rate: $(10-1) \div 10 \times 100\% = 90\%$
- Modern processor usually has more than 95% branch prediction rate.

Practice

Consider three branch prediction schemes: predict not taken, predict taken, and dynamic prediction. Assume that they all have zero penalty when they predict correctly and two cycles when they are wrong. Assume that the average predict accuracy of the dynamic predictor is 90%. Which predictor is the best choice for the following branches? Calculate the CPI of three branch prediction schemes.

- A conditional branch that is taken with 5% frequency
  - Static not taken: $\text{CPI} = 95\% \times 1 + 5\% \times 3 = 1.1$
- A conditional branch that is taken with 95% frequency
  - Static taken: $\text{CPI} = 95\% \times 1 + 5\% \times 3 = 1.1$
- A conditional branch that is taken with 70% frequency
  - Static not taken: $\text{CPI} = 30\% \times 1 + 70\% \times 3 = 2.4$
  - Static taken: $\text{CPI} = 70\% \times 1 + 30\% \times 3 = 1.6$
  - Dynamic predictor: $\text{CPI} = 90\% \times 1 + 10\% \times 3 = 1.2$

Exception

Exceptions

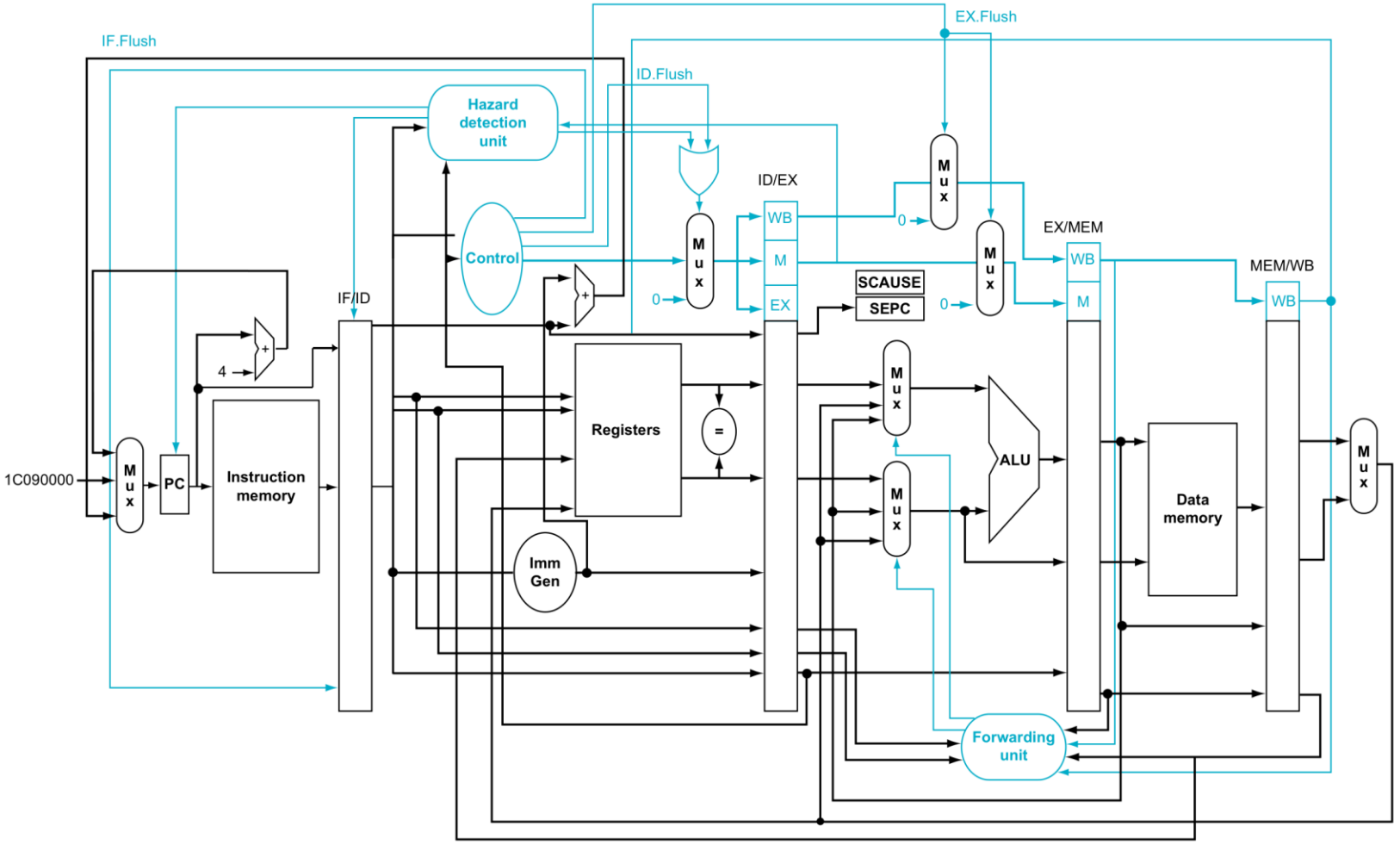
- Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow.
  - Trying to execute an undefined instruction.
  - An I/O device request.
  - An OS service request (e.g., a page fault, TLB exception).
  - A hardware malfunction.
- The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code).
- The software (OS) looks at the cause of the exception and deals with it.

Two Types of Exceptions

- Interrupts – asynchronous to program execution
  - caused by external events.
  - may be handled between instructions, so can let the instructions currently active in the pipeline complete before passing control to the OS interrupt handler.
  - simply suspend and resume user program.
- Traps (Exception) – synchronous to program execution
  - caused by internal events.
  - condition must be remedied by the trap handler for that instruction, so must stop the offending instruction midstream in the pipeline and pass control to the OS trap handler.
  - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted.
- In RISC-V, hardware stores the exception PC and cause in registers for OS to handle:
  - SEPC: A 64-bit register used to hold the address of the affected instruction.
  - SCAUSE: A register used to record the cause of the exception. In the RISC-V, this register is 64 bits, although most bits are currently unused.

Final Datapath

- Added ID.Flush and EX.Flush.
- Added the OS exceptional handler PC as input.



Summary

Summary

- All modern processors use pipelining for performance
- Pipeline clock rate limited by slowest stage – so designing a balanced pipeline is important
- Must detect and resolve hazards
  - Structural hazards – resolved by designing the pipeline correctly and adding resources
  - Data hazards
    - Stall (impacts CPI)
    - Forward (requires hardware support)
  - Control hazards – resolve the branch in as early a stage in the pipeline
    - Stall (impacts CPI)
    - Static and dynamic prediction (requires hardware support)
- Pipelining complicates exception handling