H0O38a Computer architectures and the HW/SW interface

Lecture 5
Pipelining Hazards

KU LEUVEN

M. Verhelst. version 2022.1

micas KU LEUVEN

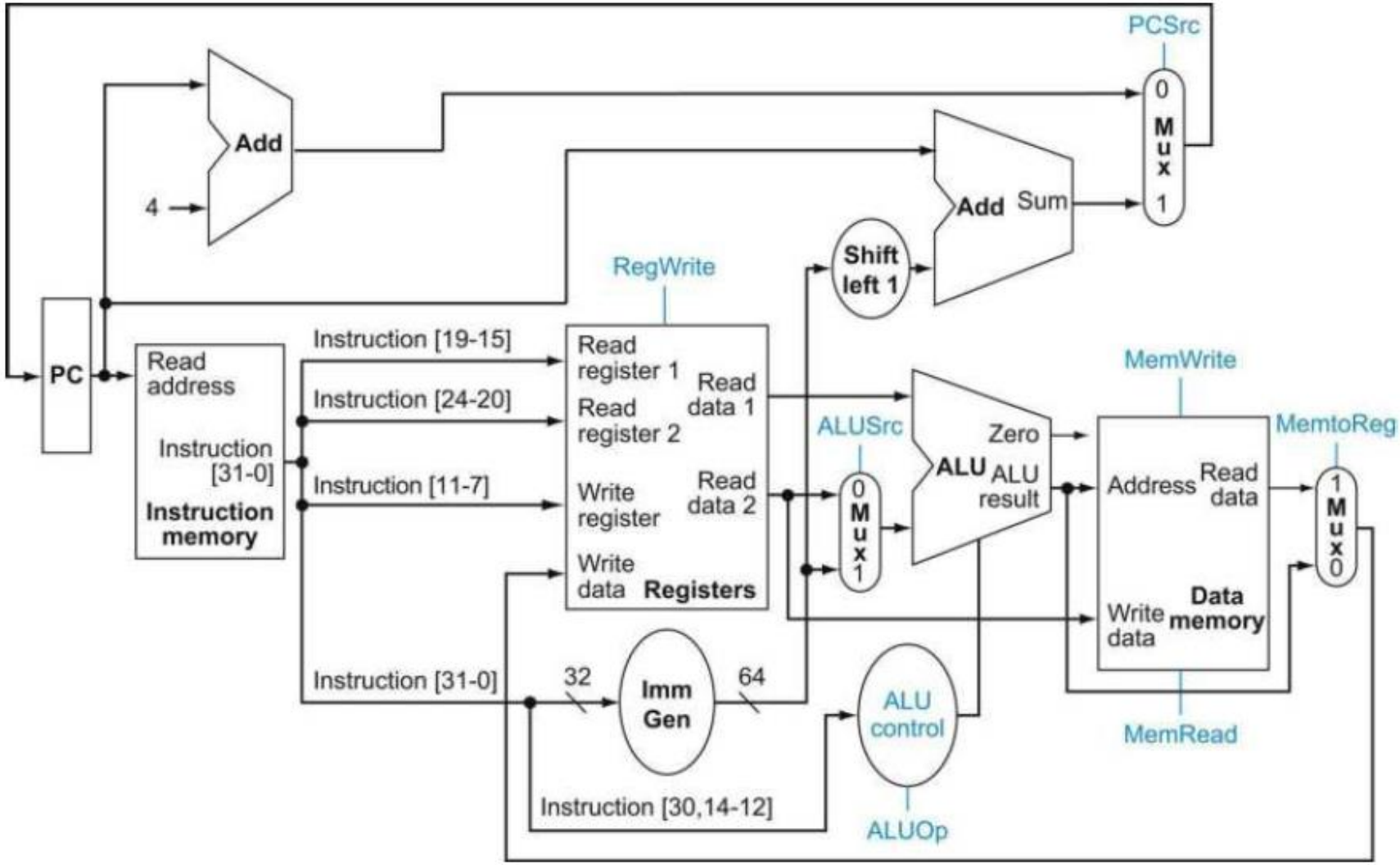
Lecture overview

- Recap last class & Ripes
- Pipelining hazards
  - Structural hazards
  - Data hazards
  - Control hazards
- Exception/interrupt handling in a pipeline

Lecture overview

- **Recap last class & Ripes**
- Pipelining hazards
  - Structural hazards
  - Data hazards
  - Control hazards
- Exception/interrupt handling in a pipeline

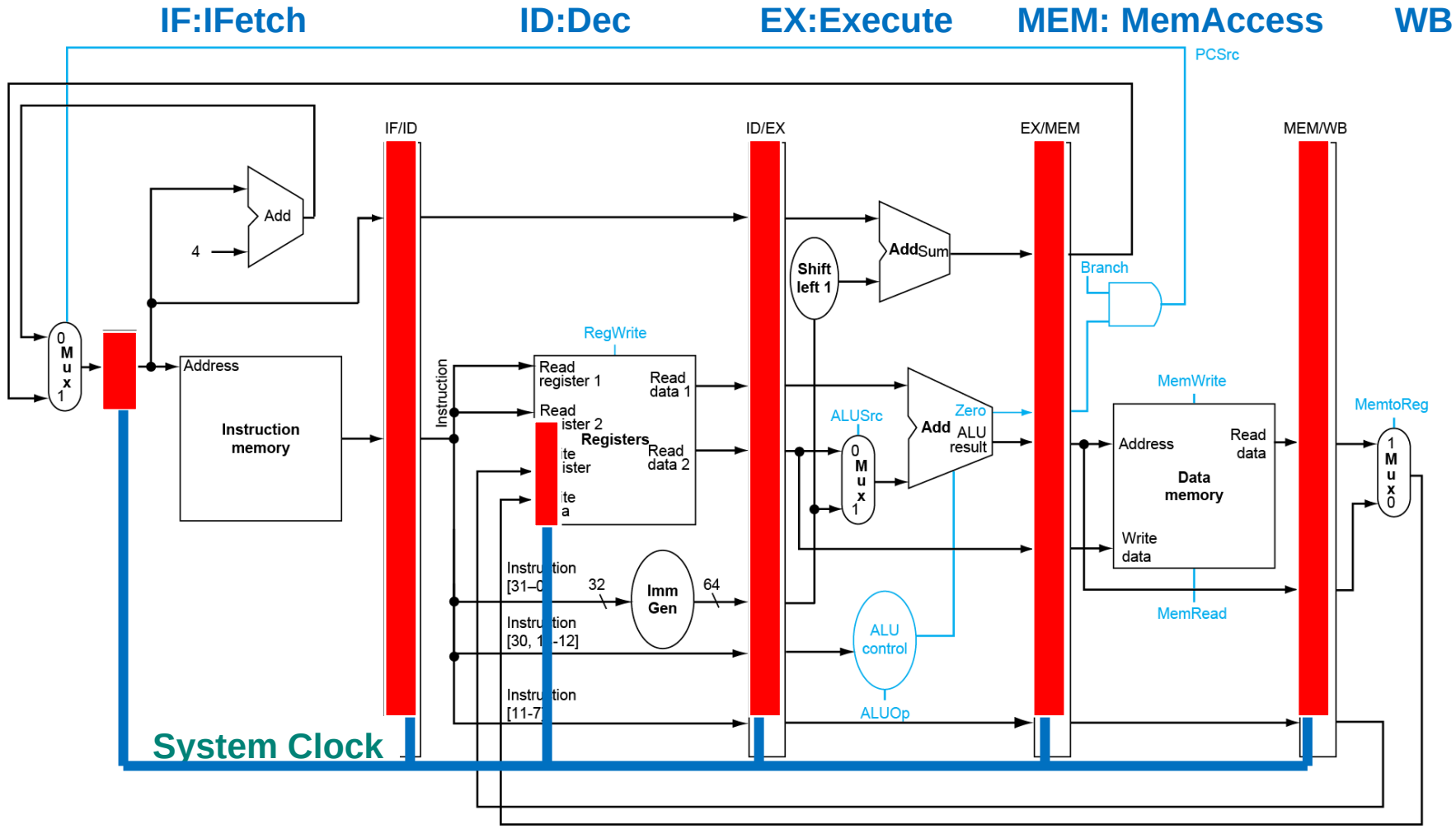
Single cycle processor



- Simple
- Can run any program
- But slow…

Processor Pipeline Datapath Additions/Modifications

- State registers between each pipeline stage to isolate them



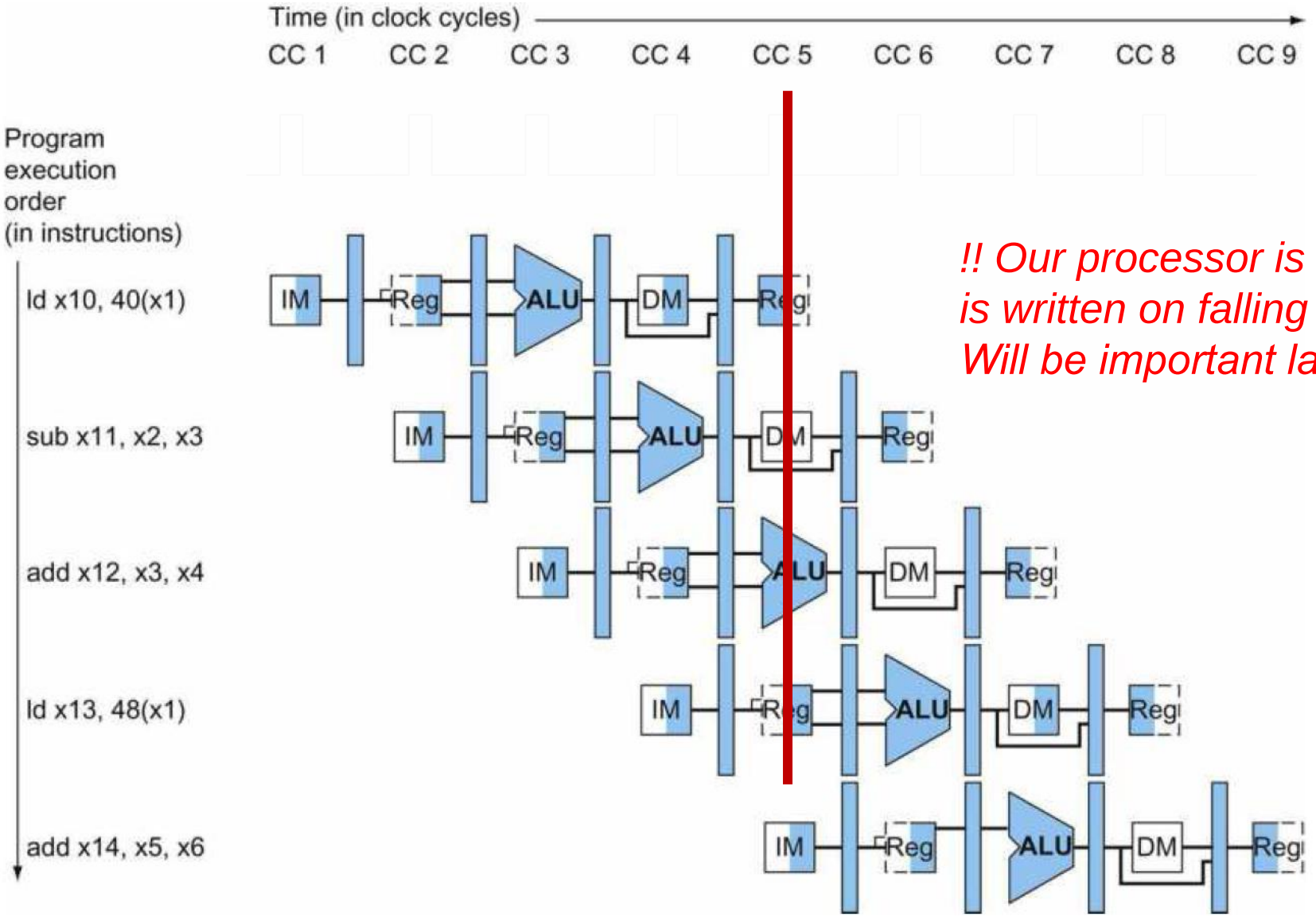
Ripes!

- Visual simulator for all processors we will discuss in this class

- Open source:
  - Install from Windows/Linux/Mac binaries: https://github.com/mortbopet/Ripes/releases
  - Or from GIT: https://github.com/mortbopet/Ripes

- Demo…

Pipelining: Always multiple instructions “in flight”



!! Our processor is such that RF is written on falling clock edge! Will be important later…

Recap Lecture 1

- Why do we pipeline?

$$\text{CPU Time} = \underbrace{\frac{\text{Instructions}}{\text{Program}}}_{\text{Instr count (IC)}} \times \underbrace{\frac{\text{Clock cycles}}{\text{Instruction}}}_{\text{CPI}} \times \underbrace{\frac{\text{Seconds}}{\text{Clock cycle}}}_{\text{Clock cycle time}}$$

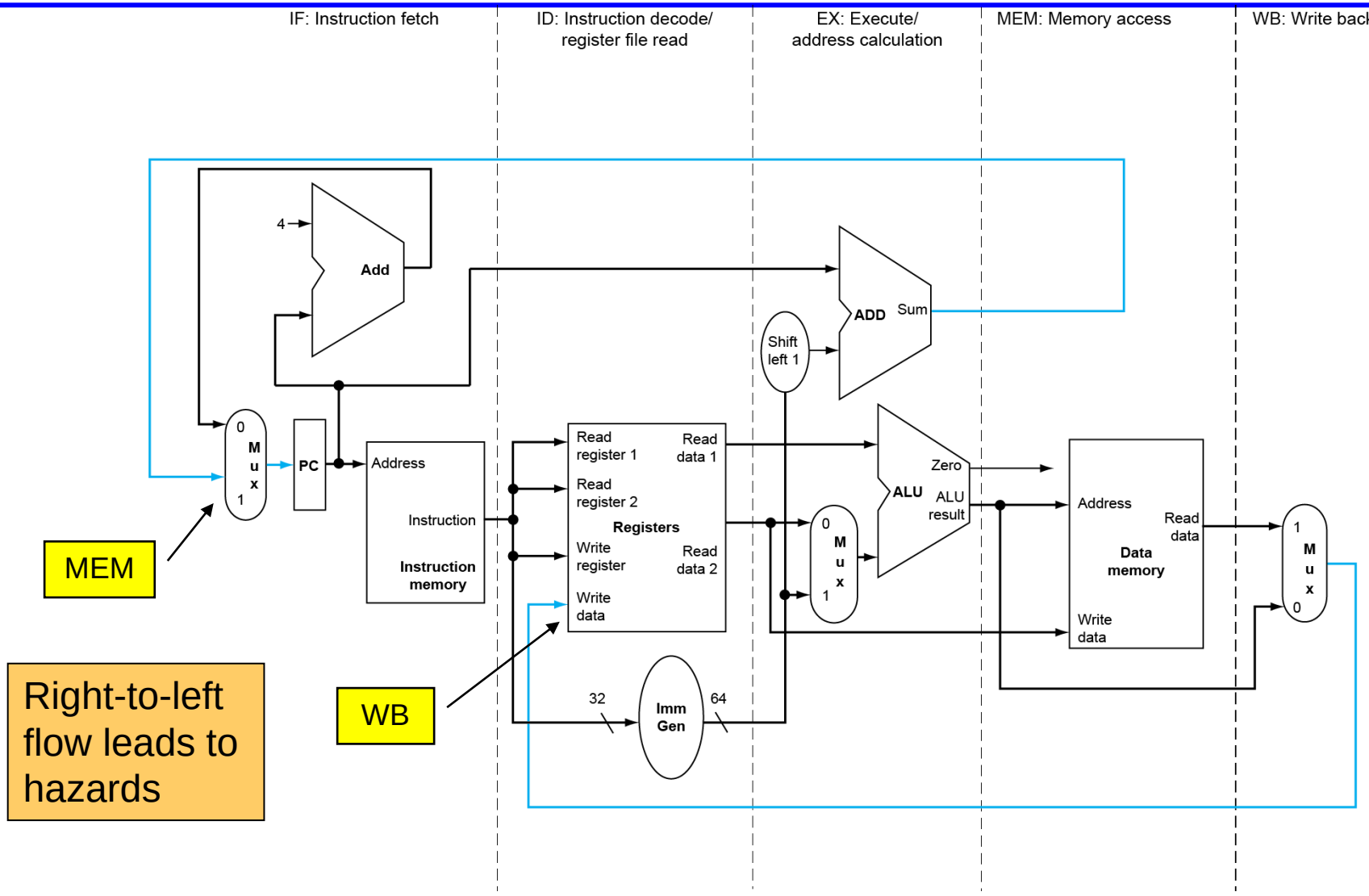
  - How much impact on clock cycle time? (optimal pipeline depth?)
  - Impact on CPI?

- Downsides of pipelining?
  - Ripes…

Lecture overview

- Recap last class & Ripes
- **Pipelining hazards**
  - Structural hazards
  - Data hazards
  - Control hazards
- Exception/interrupt handling in a pipeline

RISC-V Pipelined Datapath

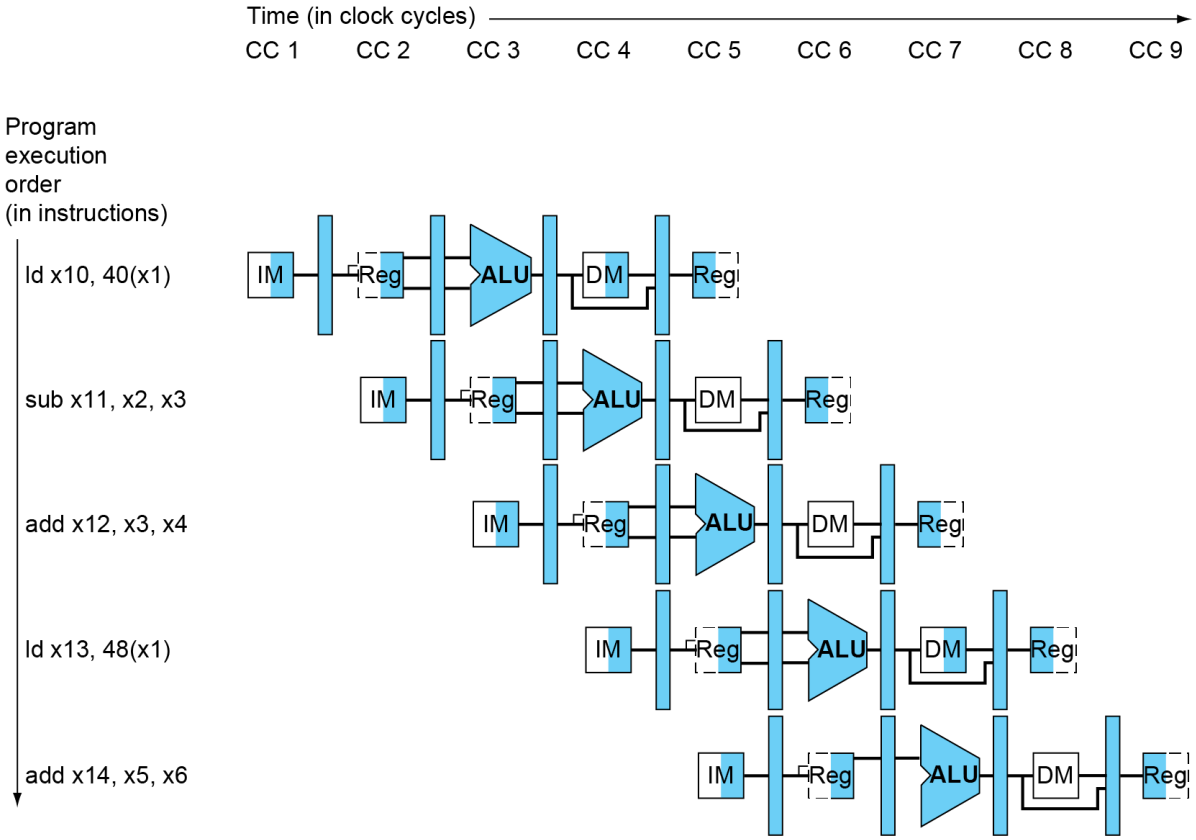


Hazards

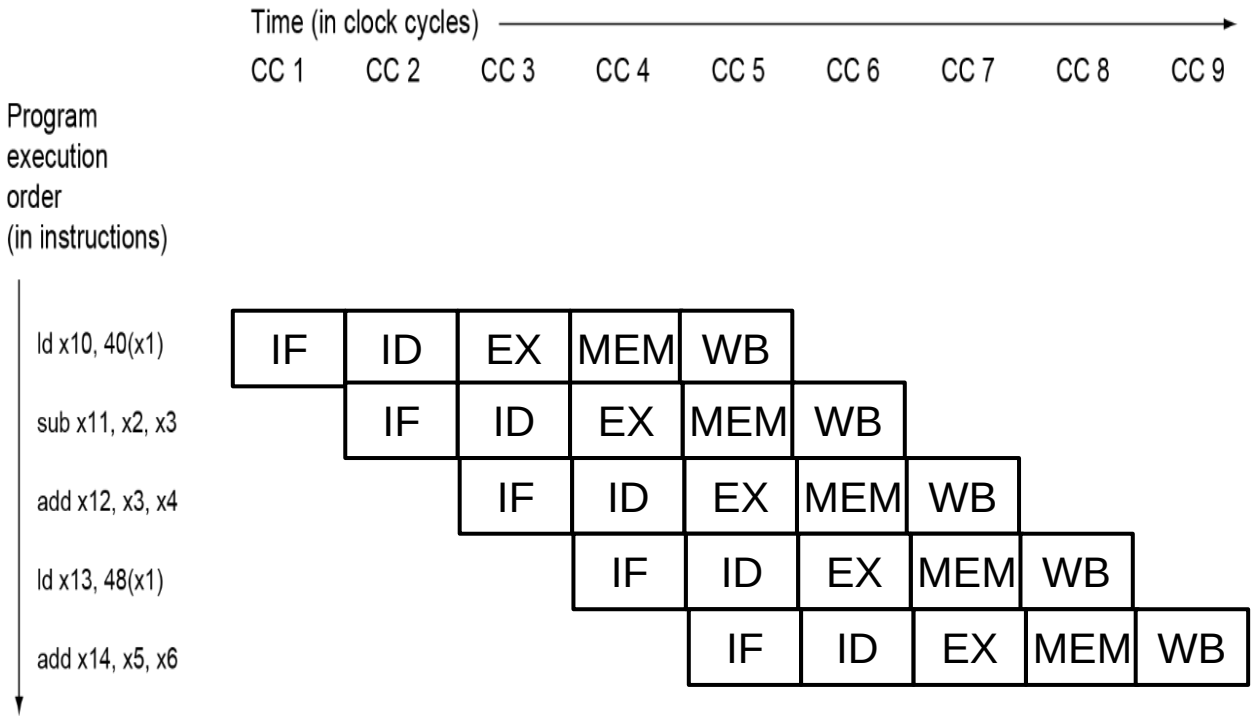
- Situations that prevent starting the next instruction in the next cycle:
  - Structure hazards
    - A required resource is busy
  - Data hazard
    - Need to wait for previous instruction to complete its data read/write
  - Control hazard
    - Deciding on control action depends on previous instruction

Multi-cycle representation to reason about pipelining

- Typically used in class



- Simplified form used in exercises, in Ripes and at exam



Lecture overview

- Recap last class & Ripes
- Pipelining hazards
  - **Structural hazards**
  - Data hazards
  - Control hazards
- Exception/interrupt handling in a pipeline

Structural Hazards

- Conflict for use of a resource
- Example 1: In RISC-V pipeline with a single memory (merging I-mem and D-mem)
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”

  ➔ Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
- Example 2: separate ALU for PC jump address calculation

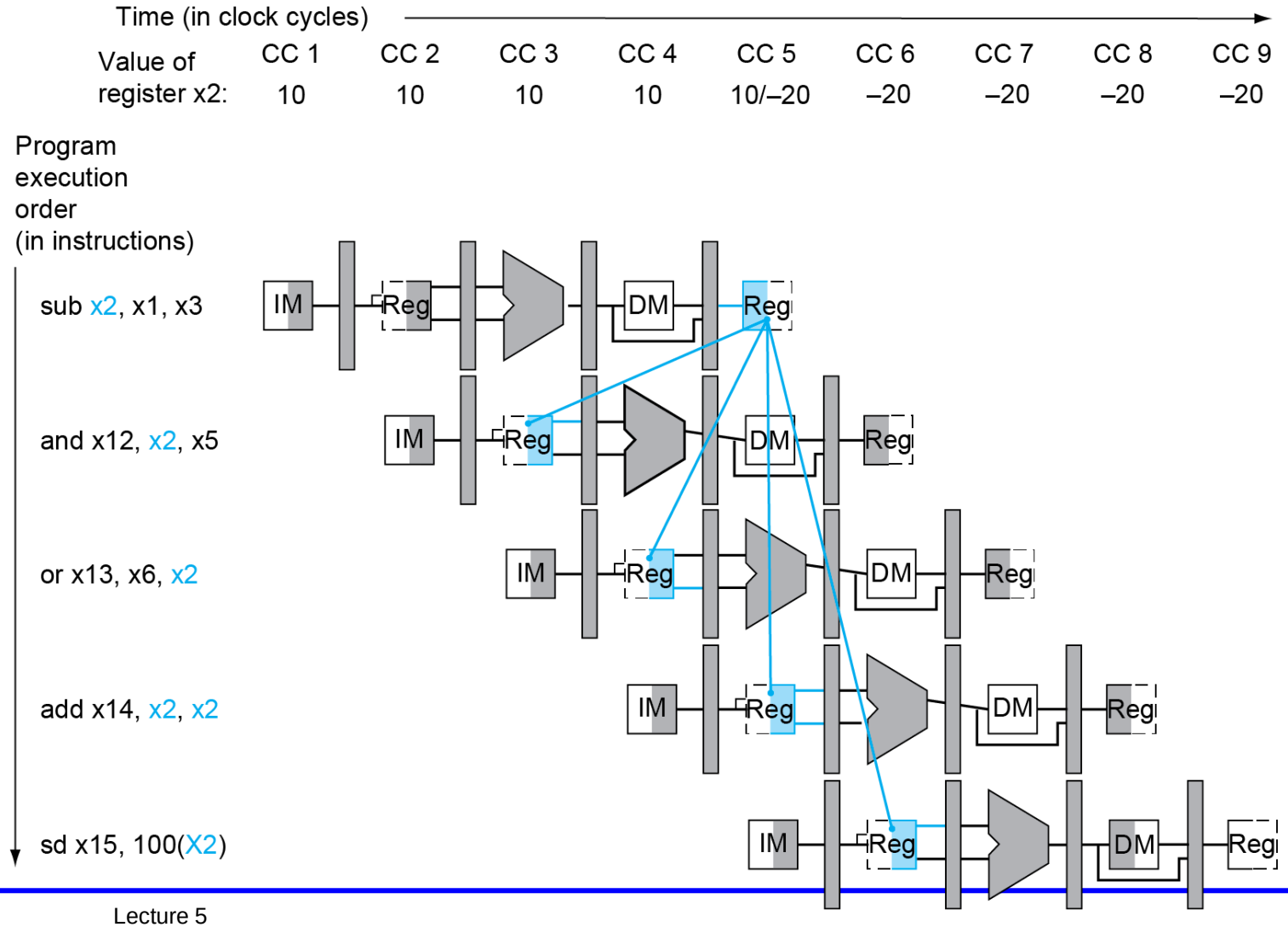
Lecture overview

- Recap last class & Ripes
- Pipelining hazards
  - Structural hazards
  - **Data hazards**
  - Control hazards
- Exception/interrupt handling in a pipeline

§4.8 Data Hazards: Forwarding vs. Stalling

Data Hazards

- An instruction depends on completion of data access by a previous instruction
- Traditional pipeline can only use data once it arrives in the RF
- Which instructions are valid / invalid in this example?
- Ripes

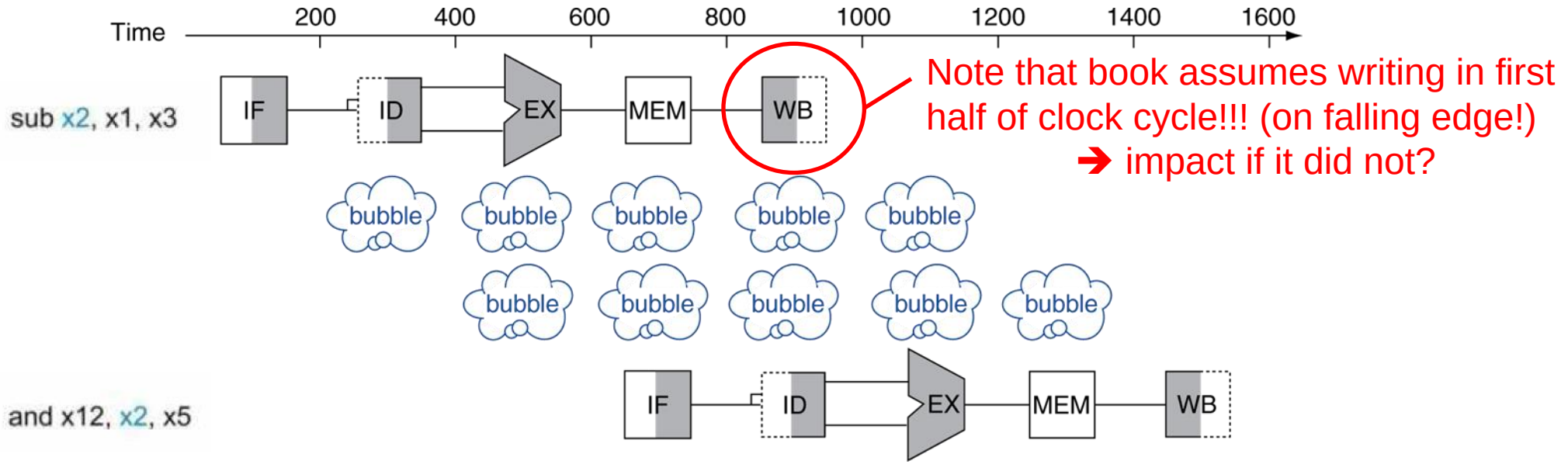


Static handling of Data Hazards

- Static = by compiler!
- How to resolve RAW?

  1.) Compiler inserts NOPs

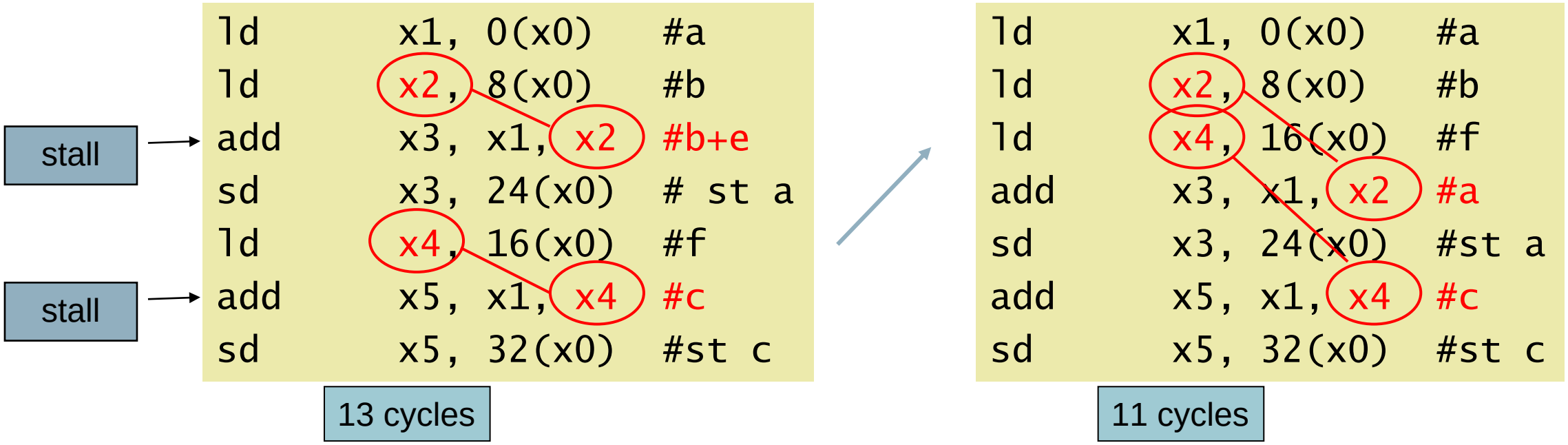
  Example of a NOP: `add x0 x0 x0` ➔ Ripes



Static handling of RAW Data Hazards

- Static = by compiler!
- How to resolve RAW?
  1.) Compiler inserts NOPs
  2.) Compiler reshuffles instructions (=code scheduling)

E.g. C code for `a = b + e; c = b + f;`

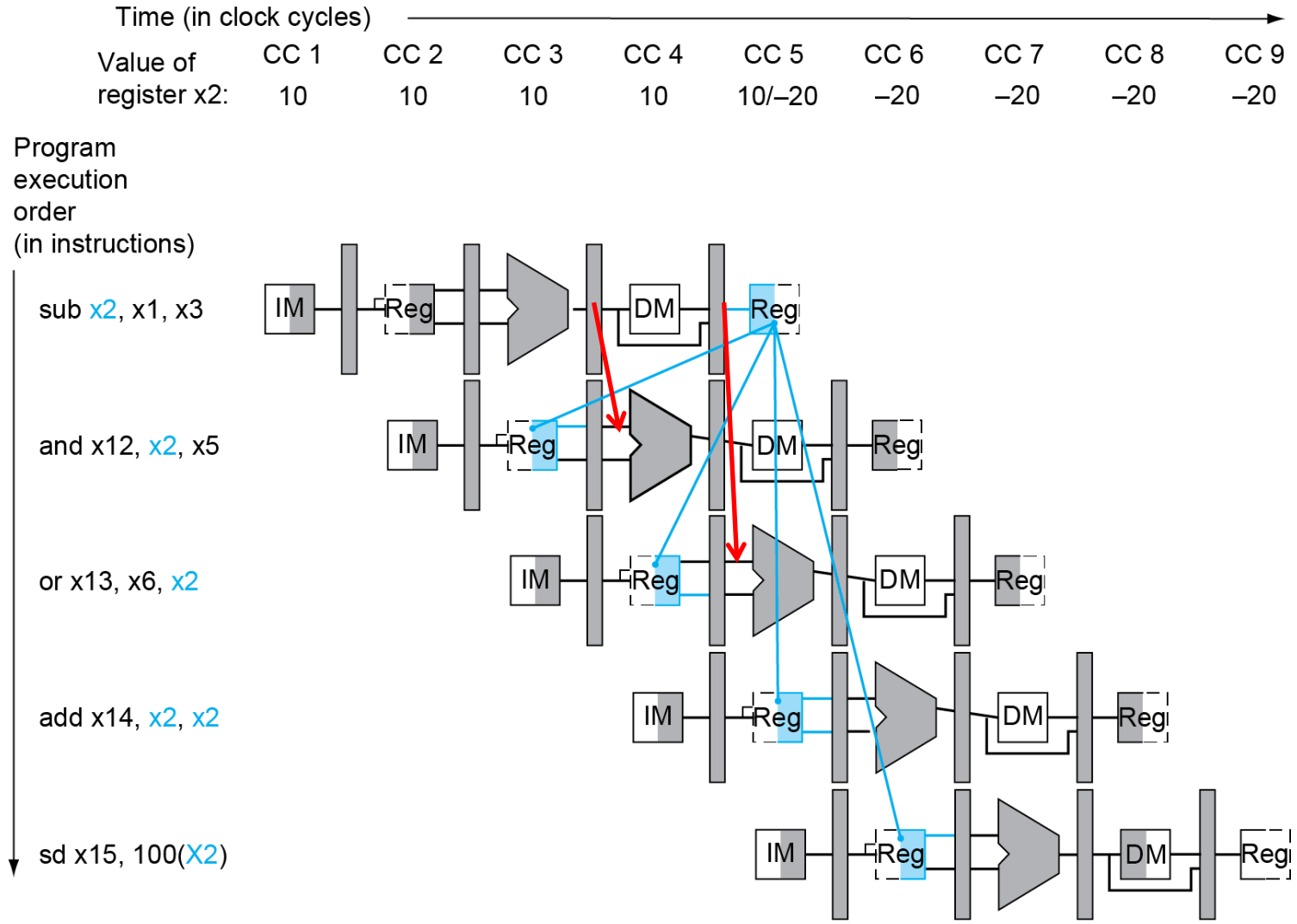


Dynamic handling of RAW hazards

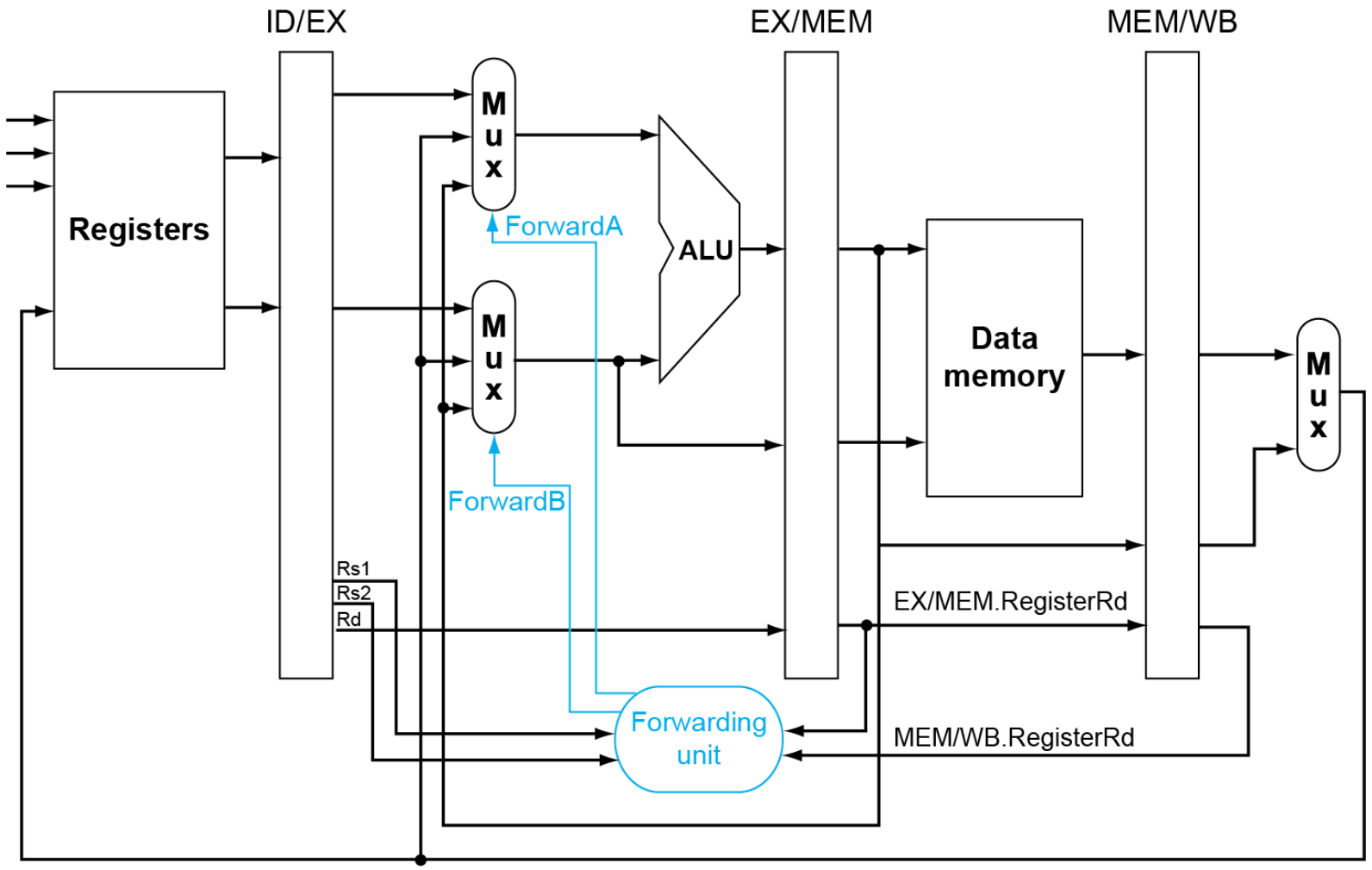
- Dynamic = online, without knowledge of compiler!

  1.) “Data bypassing” aka “forwarding”

  =Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath



Forwarding Paths



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs1 #register number for Rs1 sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- Data hazards when

  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

  → Fwd from EX/MEM pipeline reg

  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

  → Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite==1, MEM/WB.RegWrite==1
- And only if Rd for that instruction is not x0
  - EX/MEM.RegisterRd ≠ x0,
    MEM/WB.RegisterRd ≠ x0

Forwarding unit?

- Continuously checks forwarding conditions for EX and MEM stage
  - E.g. For EX hazard:

```
If (EX/MEM.RegWrite ==1
and EX/MEM.RegisterRd != x0
and EX/MEM.RegisterRd == ID/EX.RegisterRs1 )
  ForwardA = 2
If (EX/MEM.RegWrite ==1
and EX/MEM.RegisterRd != x0
and EX/MEM.RegisterRd == ID/EX.RegisterRs2 )
  ForwardB = 2
```

- (cfr. For MEM/WB hazard) ➔ try it!

Double Data Hazard

- Consider the sequence:

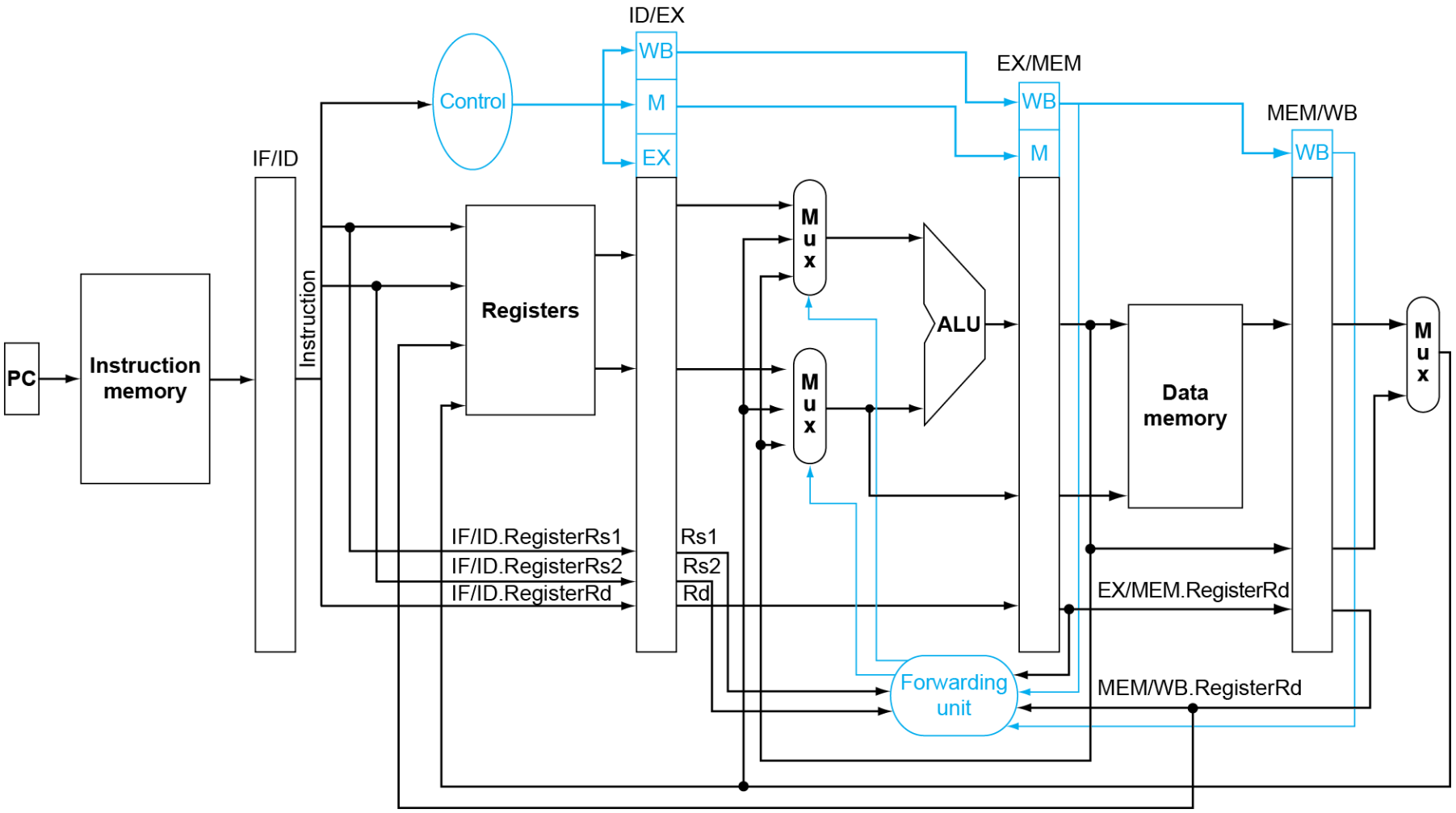
```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

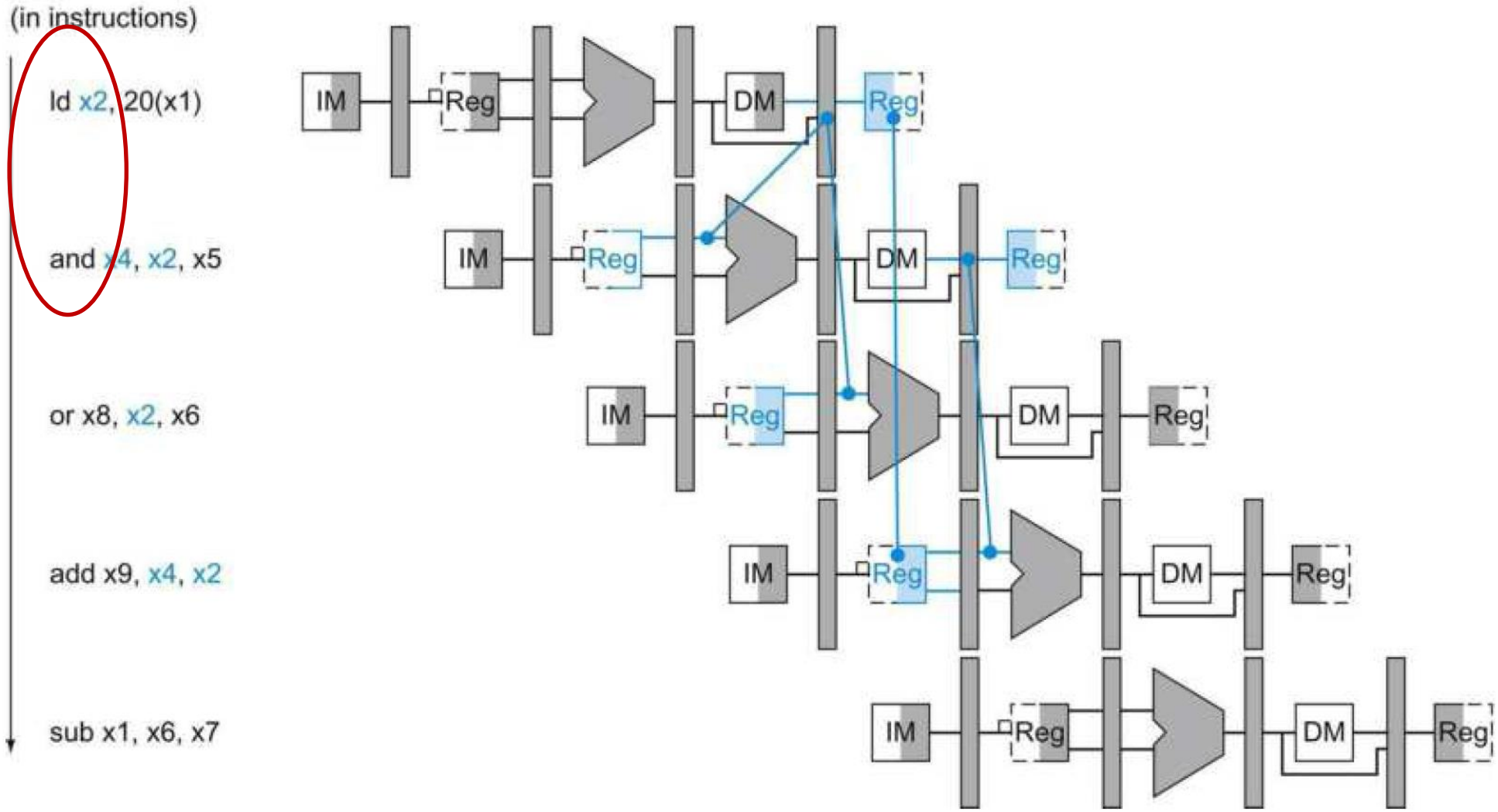
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd != x0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != x0)
      and (EX/MEM.RegisterRd == ID/EX.RegisterRs1))
      ForwardA = 1
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd != x0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != x0)
      and (EX/MEM.RegisterRd == ID/EX.RegisterRs2))
      ForwardB = 1

Datapath with Forwarding (+ control signals)

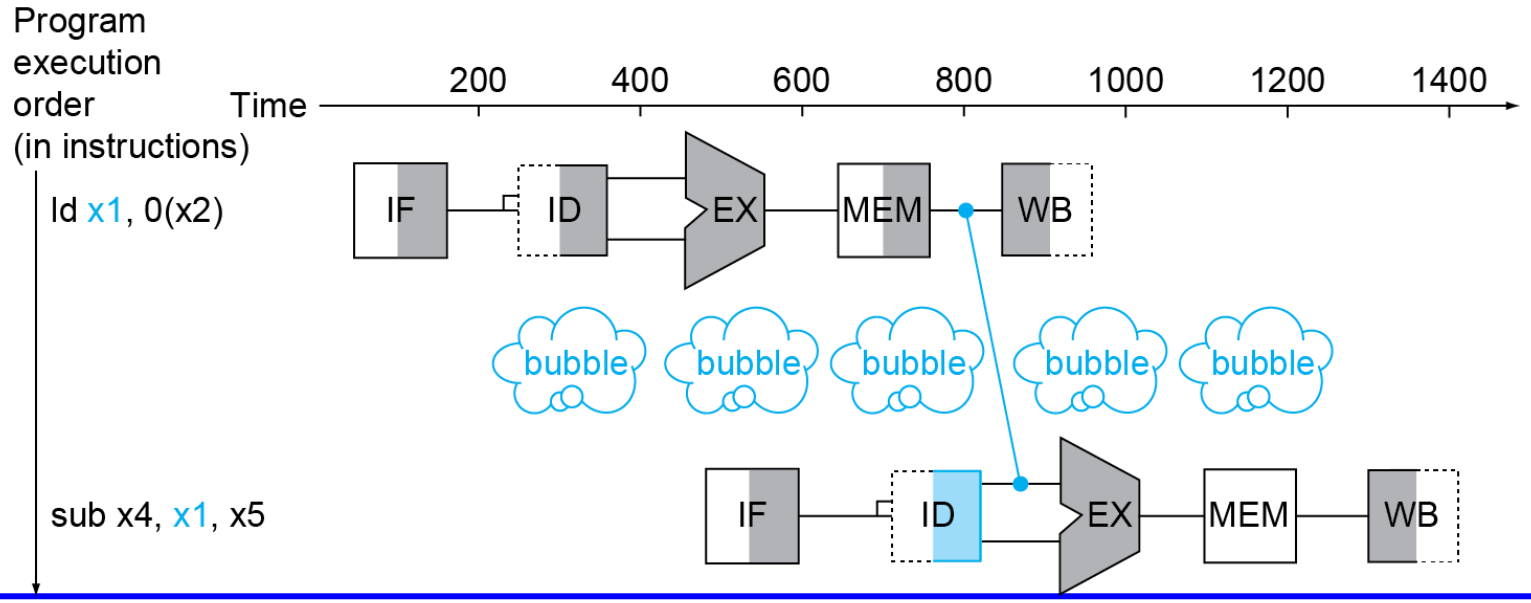


Forwarding with load-use?

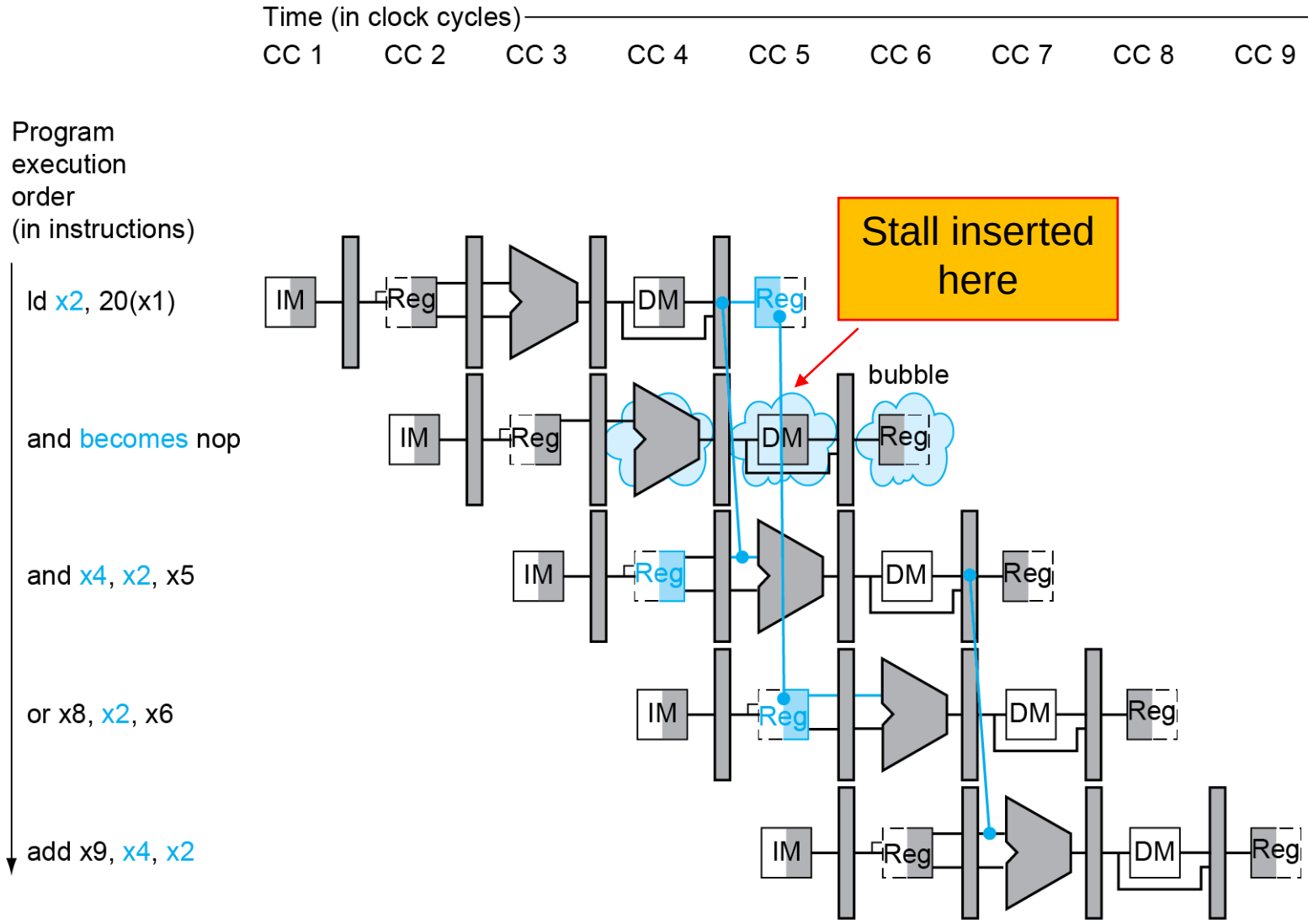


Dynamic handling of RAW hazards

- Dynamic = online, without knowledge of compiler!
  1.) “Data bypassing” aka “forwarding”
  - Can’t always avoid stalls by forwarding: Can’t forward backward in time!
  2.) **If needed, stall** pipeline, using dynamic data dependency resolution
  - Used for *load-use*
  - Also used for control hazards



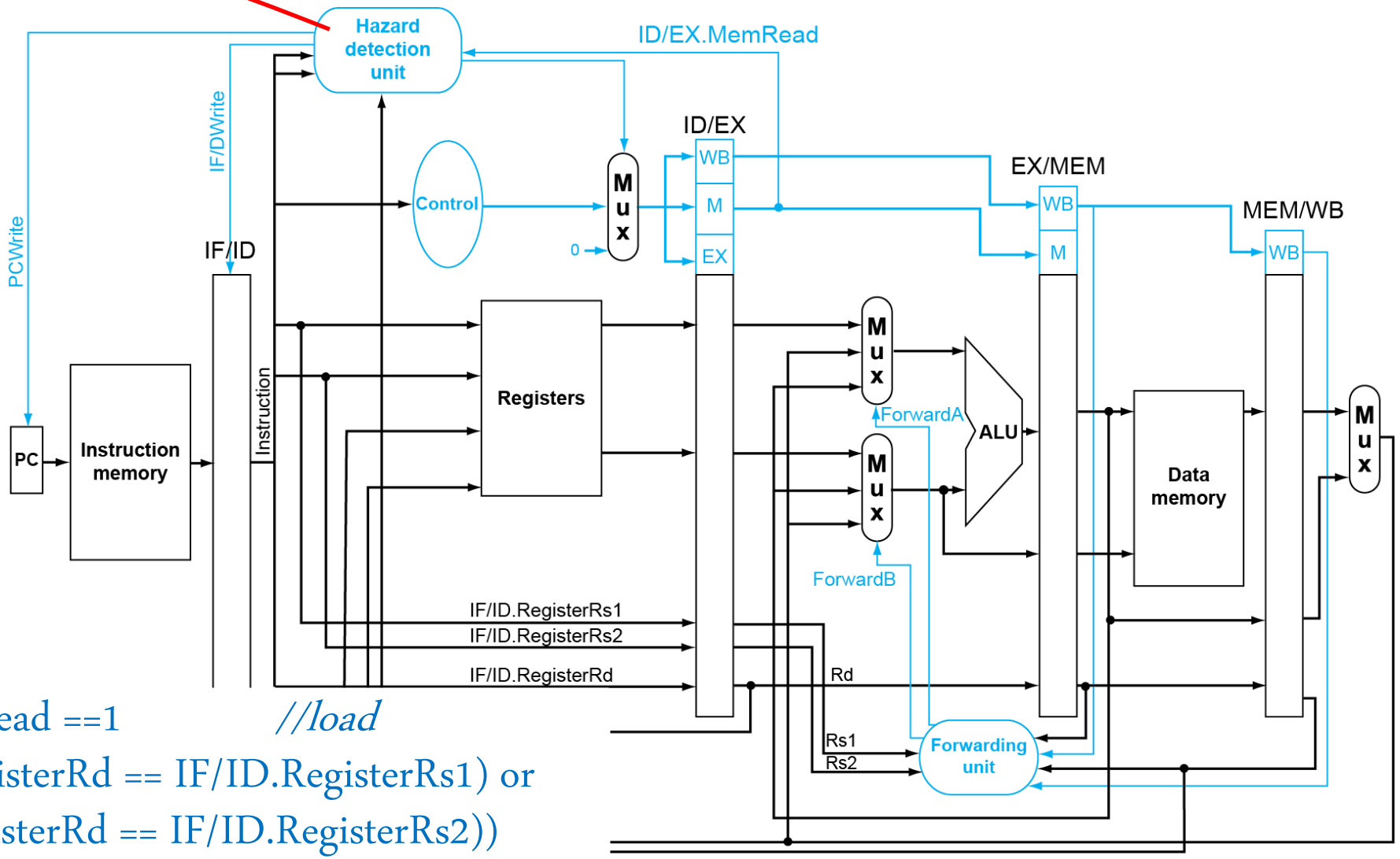
Target: stall the pipeline!



How?

Don't change the PC, just don't fetch a new instruction and insert a "0"-instruction
= "stalling" the pipeline

Datapath with Hazard Detection



If (ID/EX.MemRead ==1 *//load*
and ((ID/EX.RegisterRd == IF/ID.RegisterRs1) or
(ID/EX.RegisterRd == IF/ID.RegisterRs2))
stall pipeline

Stalls and Performance

The BIG Picture

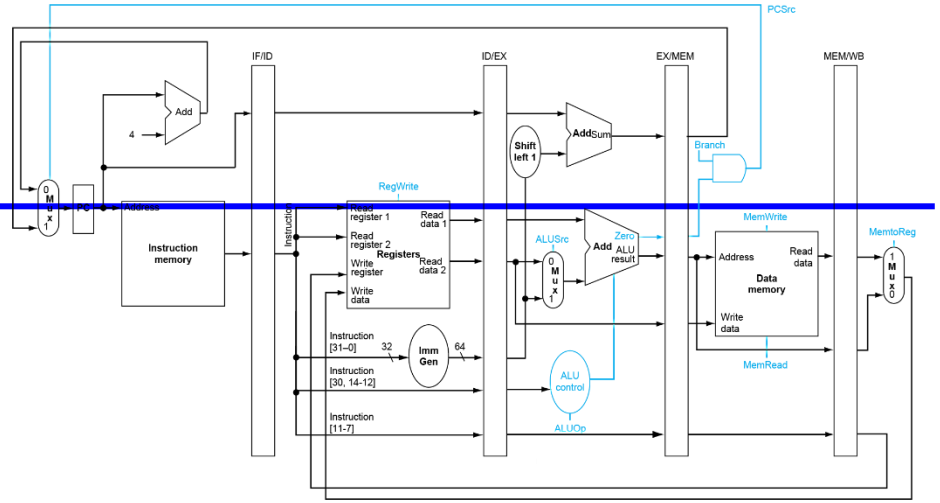
- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Lecture overview

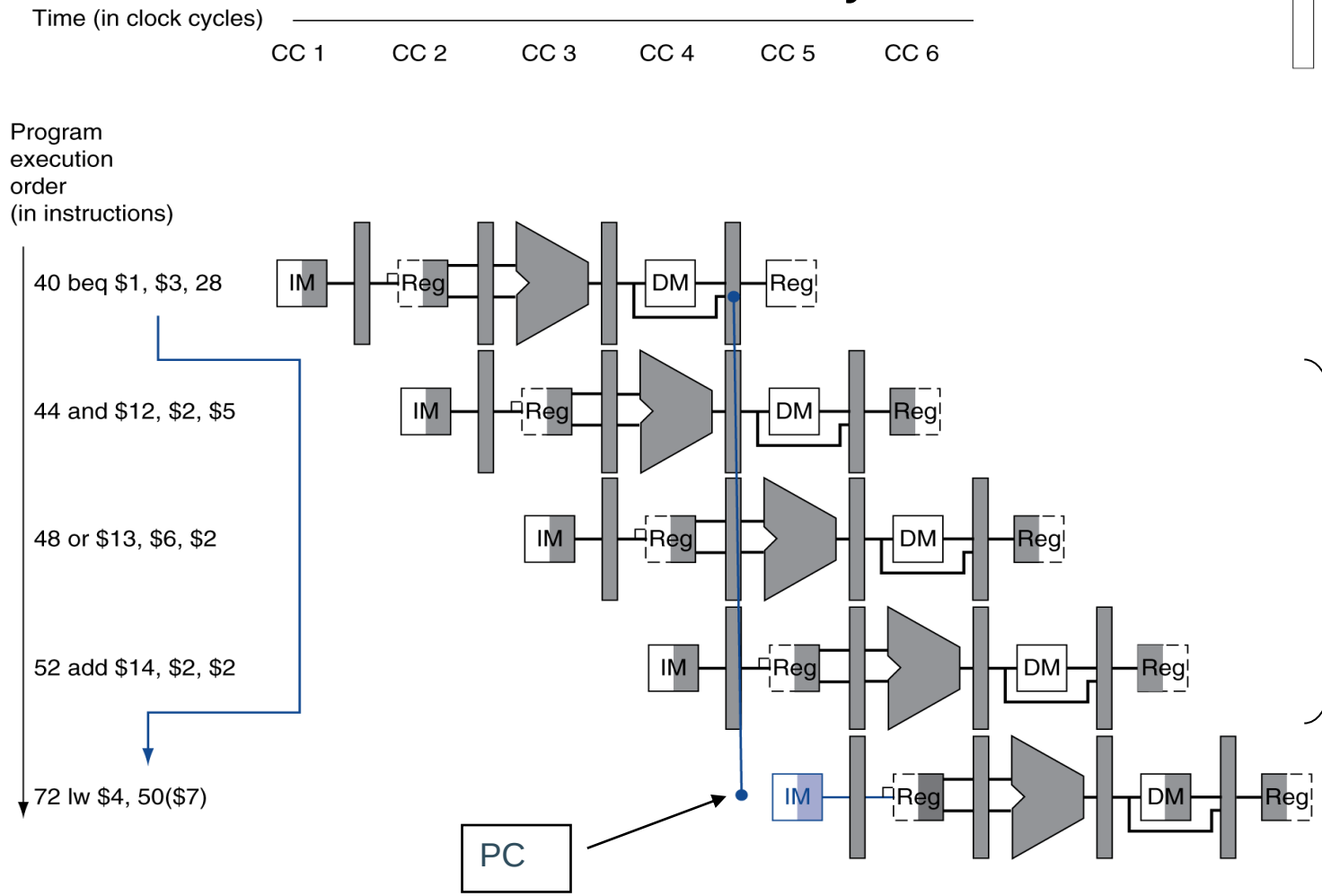
- Recap last class & Ripes
- Pipelining hazards
  - Structural hazards
  - Data hazards
  - **Control hazards**
- Exception/interrupt handling in a pipeline

§4.9 Control Hazards

Control Hazards



- Fetching next instruction depends on branch outcome, which is not known yet



Ripes…

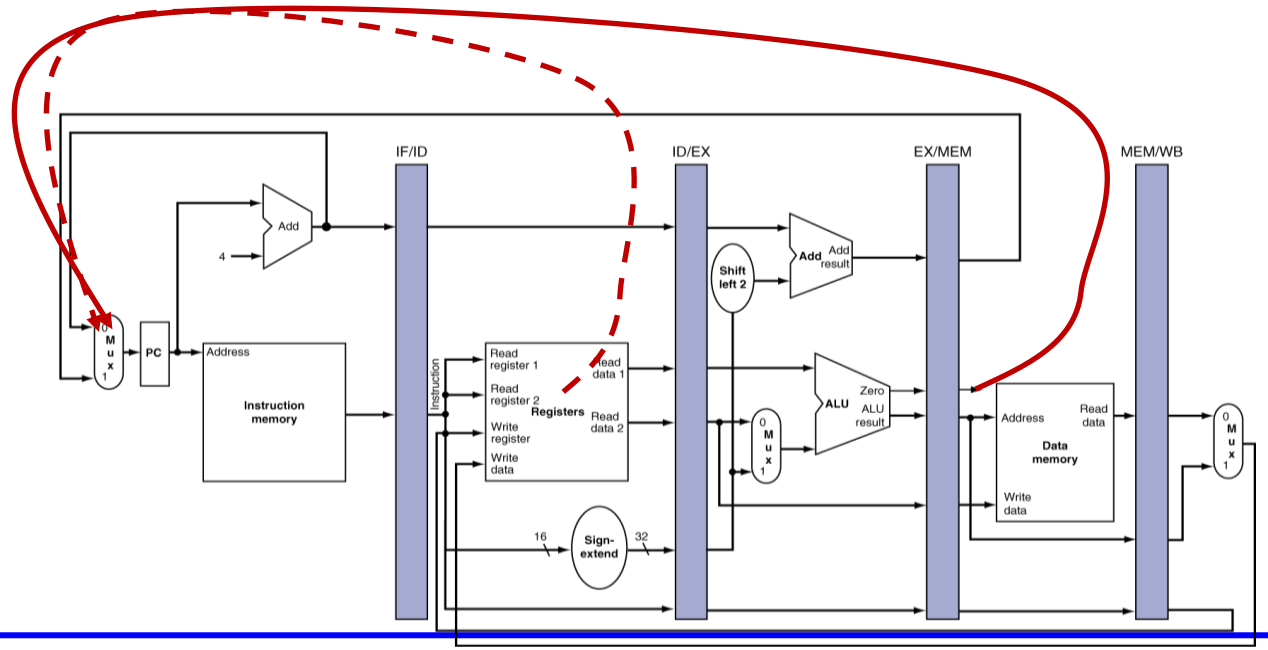
Wrong instructions fetched & executed!!!

3 cycles

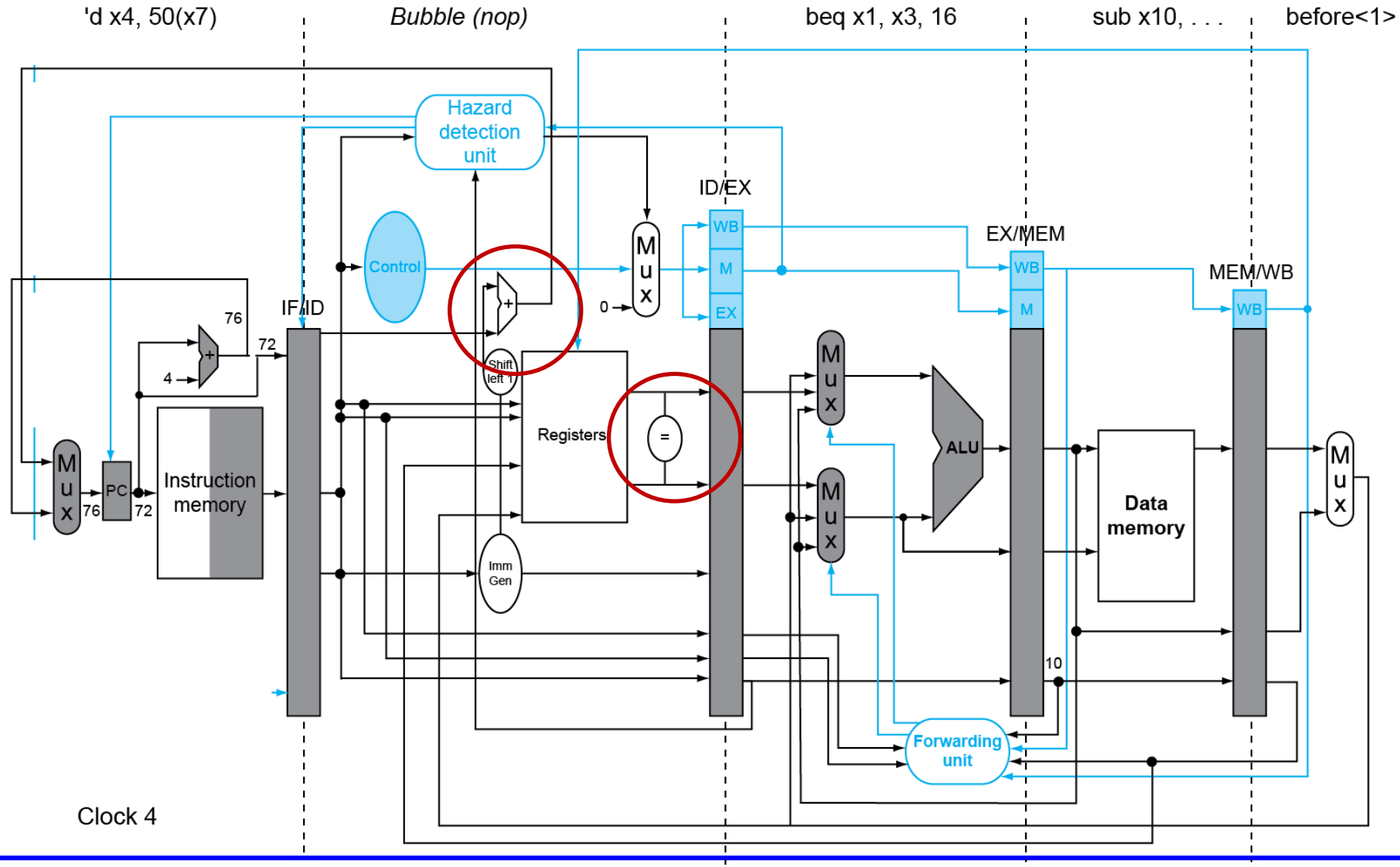
Static handling of Control Hazards

Done by the compiler, up front!

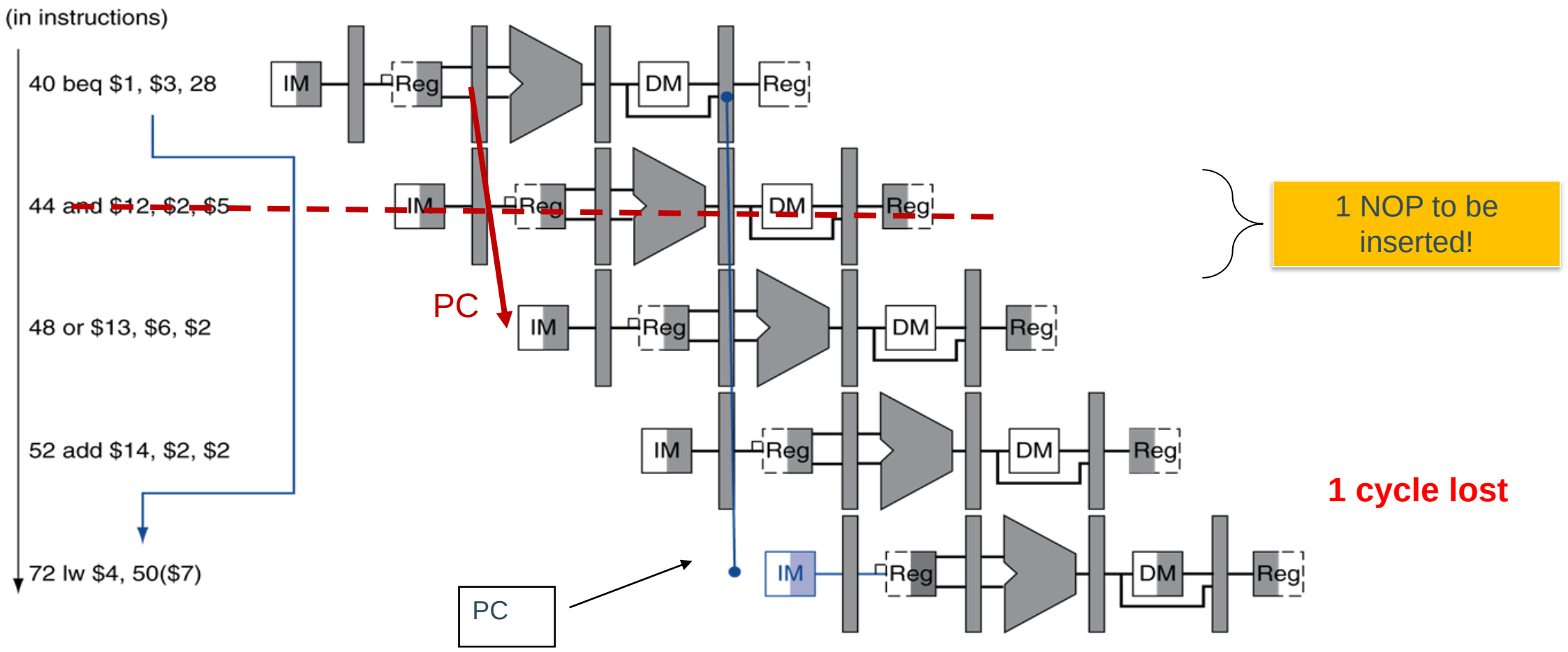
1. Always insert 3 NOPs (bubbles) after a branch
   - Ripes…
   - Can be reduced to 1 bubble with hardware architecture change:
     moving “eq. detection” to ID stage ➔ 1-stage bubble



Address compute now in ID stage



Move EQ checking...



Static handling of Control Hazards

Done by the compiler, up front!

1. Always insert 3 NOPs (bubbles) after a branch
   - Can be reduced to 1 bubble with hardware change:
     moving “eq. detection” to ID stage ➔ 1 bubble
2. Loop unrolling

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead

- Use different registers per replication
  - Called “**register renaming**”
  - To avoid over-writing, or false dependencies
  - **Compiler** does this (offline)!

Loop unrolling example

```
lp:     lw      x4,0(x1)    # x4=array element
        add     x4,x4,x2    # add x2 to x4
        sw      x4,0(x1)    # store result
        addi    x1,x1,-4    # decrement pointer
        bne     x1,$0,lp    # branch if x1 != 0
```

```
lp:     lw      x4,0(x1)      # x4=array element
        lw      x5,-4(x1)     # x5=next array element
        add     x4,x4,x2      # add x2 to x4
        add     x5,x5,x2      # add x2 to x5
        sw      x4,0(x1)      # store result x4
        sw      x5,-4(x1)     # store result x5
        addi    x1,x1,-8      # decrement pointer
        bne     x1,$0,lp      # branch if x1 != 0
```

E.g. unroll loop with factor 2
➔ 2 times less branch operations
(in general for loop with ***n*** iterations: make ***k*** copies of loop body and execute ***n/k*** times)

Increases basic block size (see further)

Note: increases nb of “live registers” vs original loop!

Static vs Dynamic handling of Control Hazards

Static techniques still result in stalls

DETERMINISTIC solutions cannot remove all stalls…

- hard for a compiler/programmer to avoid all loops and stalls…
- Cost grows if pipeline gets deeper and wider (e.g. VLIW) and deeper…

➔ Non-deterministic branch handling methods

Dynamic (run-time) techniques can further reduce this (at the expense of determinism)

Deterministic vs Non-deterministic branch handling

- Deterministic methods ➔ *know…*
  - Always same number of cycles
  - Known on beforehand, managed by the compiler
  - More cycles on average

  (for hard-real-time systems)

- Non-deterministic methods ➔ *predict and flush if misprediction...*
  - On average more efficient (less cycles)
  - Managed at run-time
  - Yet unpredictable (can be many more)

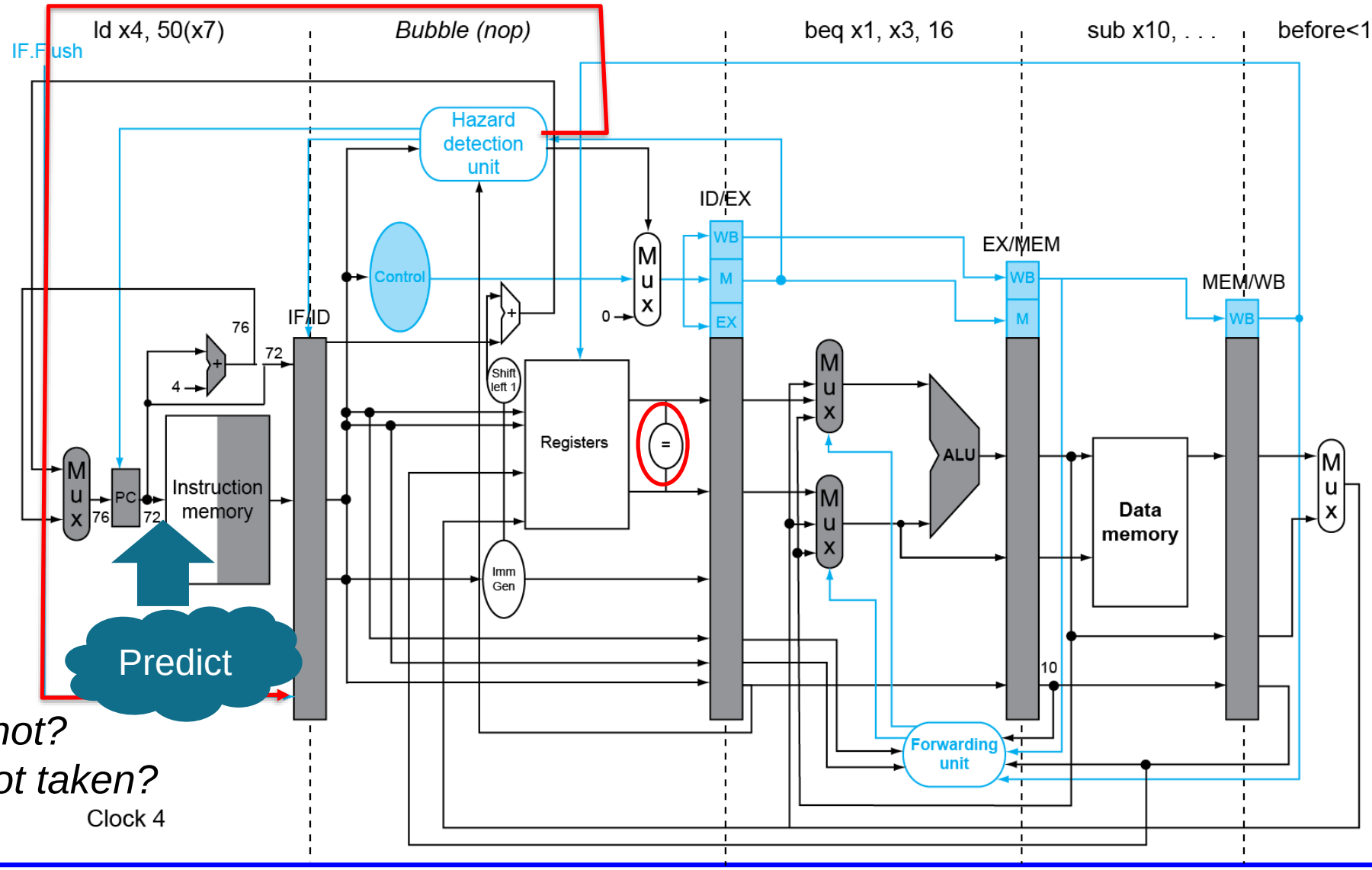
  (for non-real-time and soft-real-time systems)

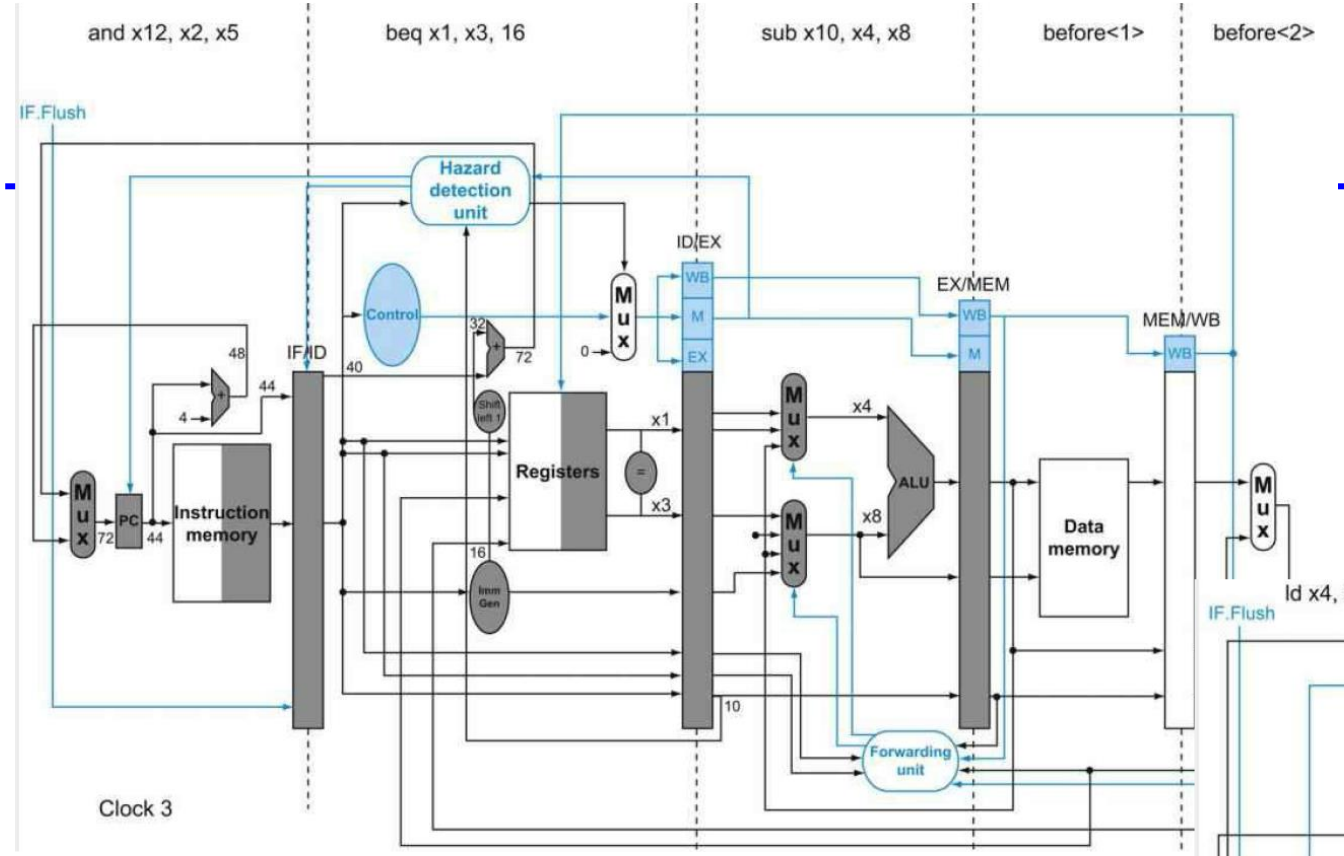
Dynamic handling of Control Hazards

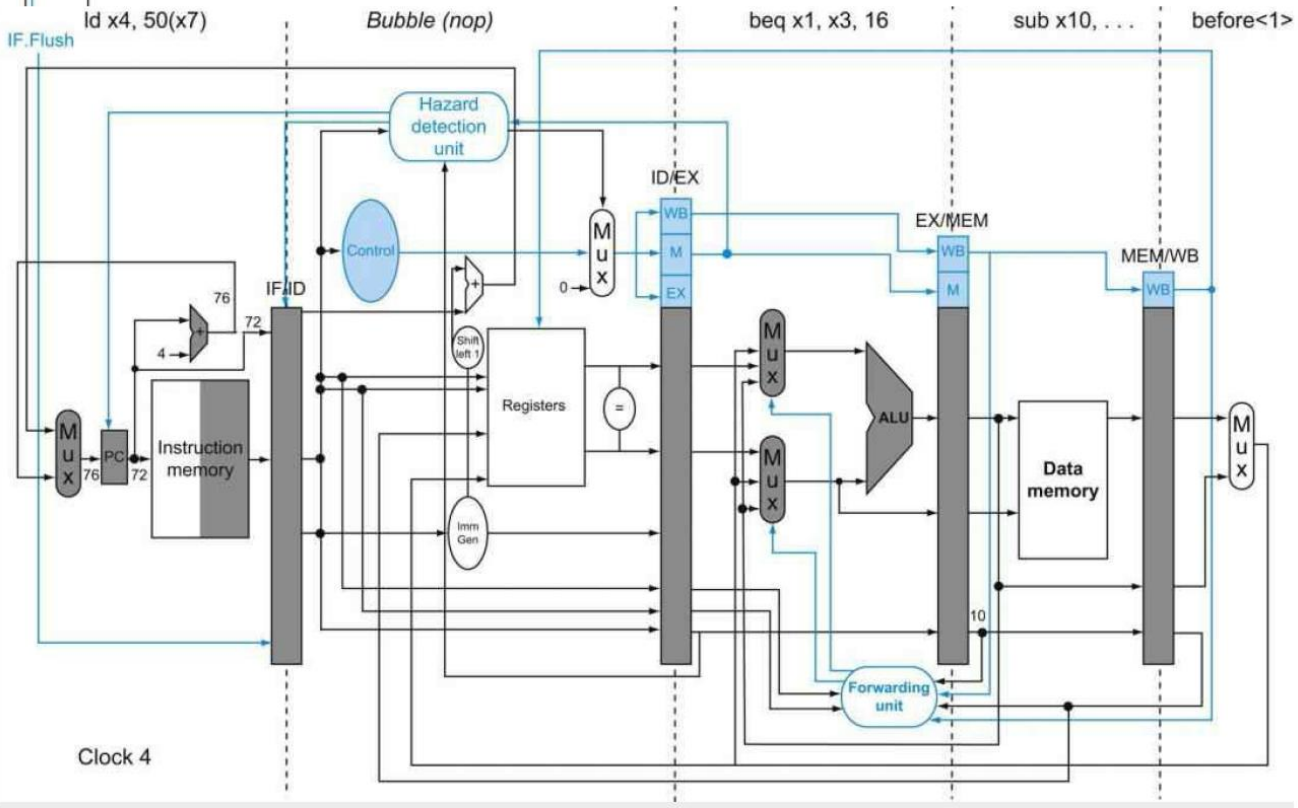
- Predict whether branch is likely to be taken or not.
- Already fetch resulting instruction in next cycle
- If prediction (later) turns out to be correct ➔ Continue
- If prediction turns out to be incorrect ➔ Correct the error = "*flush the pipeline*"
  - Remove all faulty executed instructions
  - (should not have changed any registers or memories yet!)
  - Restart from correct instruction

How do we flush?

Pipeline flushing




and x12, x2, x5
beq x1, x3, 16
sub x10, x4, x8
before<1>
before<2>
IF.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
IF/ID
Registers
Instruction memory
PC
ALU
Data memory
Forwarding unit
Imm Gen
Shift left 1
Mux
x1
x3
x4
x8
Clock 3


ld x4, 50(x7)
Bubble (nop)
beq x1, x3, 16
sub x10, . . .
before<1>
IF.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
WB
M
EX
IF/ID
Registers
Instruction memory
PC
ALU
Data memory
Forwarding unit
Imm Gen
Shift left 1
Mux
Clock 4

Branch prediction schemes

1. Simple prediction ➔ E.g. predict not taken…
   - Low accuracy: ~30-40%

2. "Static" prediction ➔ forward: not taken; backward: taken
   - Better accuracy: ~60-70%
   - Why?
     - FW branches ➔ exceptions
     - BW branches ➔ iterative loops (e.g. for i=1..100)

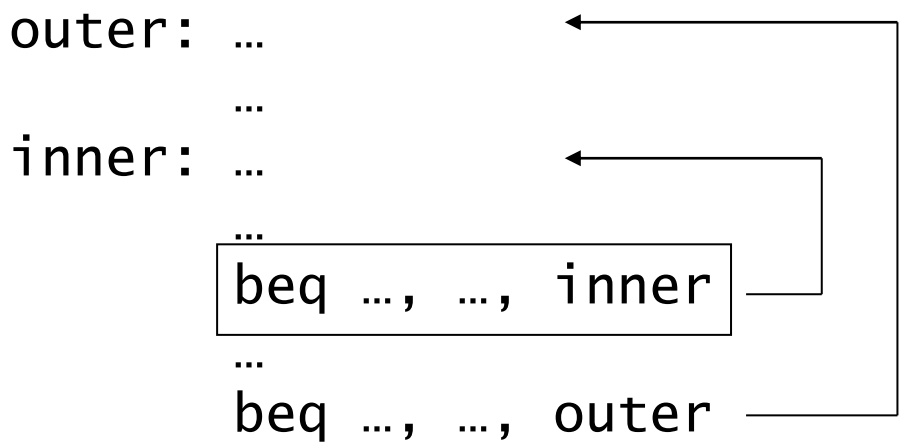
Branch prediction schemes (2)

1. Simple prediction ➔ E.g. predict not taken…
2. “Static” prediction ➔ forward: not taken; backward: taken
3. “Dynamic” prediction ➔ Use run-time information
   - Keep a record whether the branch was taken in the past or not

➔ Branch history table (also call “branch prediction buffer”)

1-Bit Predictor

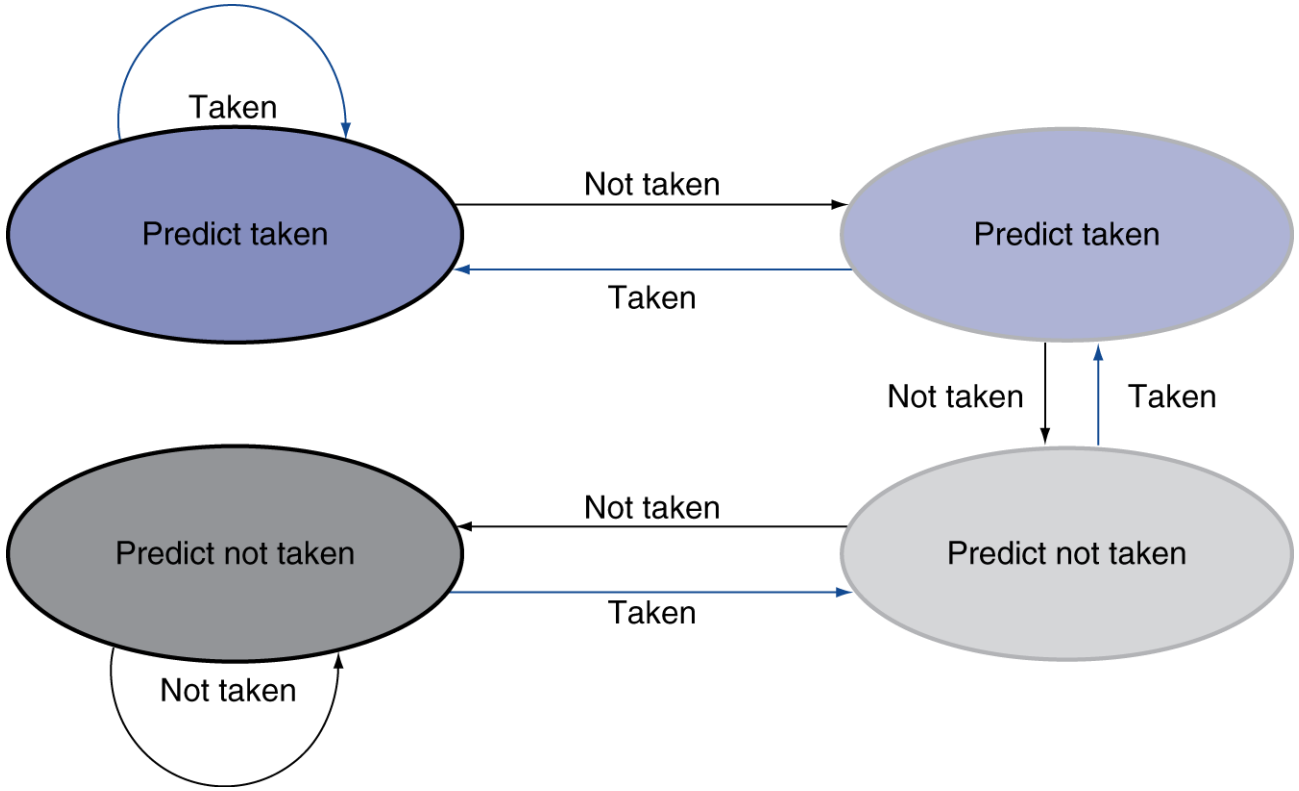
- Predict same as last time this branch was visited
- But… Inner loop branches mispredicted twice!



  - Mispredict on last iteration of inner loop
    - Predict taken instead of not taken
  - Then mispredict on first iteration of inner loop next time around
    - Predict not taken instead of taken

2-Bit Predictor

- Only change prediction on two successive mispredictions



Lecture overview

- Recap last class & Ripes
- Pipelining hazards
  - Structural hazards
  - Data hazards
  - Control hazards
- **Exception/interrupt handling in a pipeline**

§4.10 Exceptions

Exceptions & interrupts

- Exceptions:
  - events other than branches or jumps that change the normal flow of instruction execution
  - *any* unexpected change in control flow
  - E.g.
    - Arithmetic overlfow
    - Executing undefined instruction

- Interrupt:
  - exception that comes from outside of the processor (cfr previous class of Prof. Lauwereins)
  - E.g.
    - I/O device request
    - Temperature monitoring alarm

What to do on interrupt/exception?

Stopping the execution of the program: *(cfr. control hazard)*

1. Let all prior instructions complete, flush all following instructions
   - Flush pipeline (now more than just the IF stage!!! Also ID and EX)
   - This flushing avoids data leading to the exception to be overwritten by faulty instructions
2. Report an error in case of exception:
   - Save address of the offending instruction in the *supervisor exception program counter* (*SEPC*)
   - Save cause of the exception in the *Supervisor Exception Cause Register (SCAUSE)*
3. Jump to a predefined sequence of instructions in OS
   - Called the “Interrupt/exception handler”
   - Check cause register / interrupt vector table content and perform action

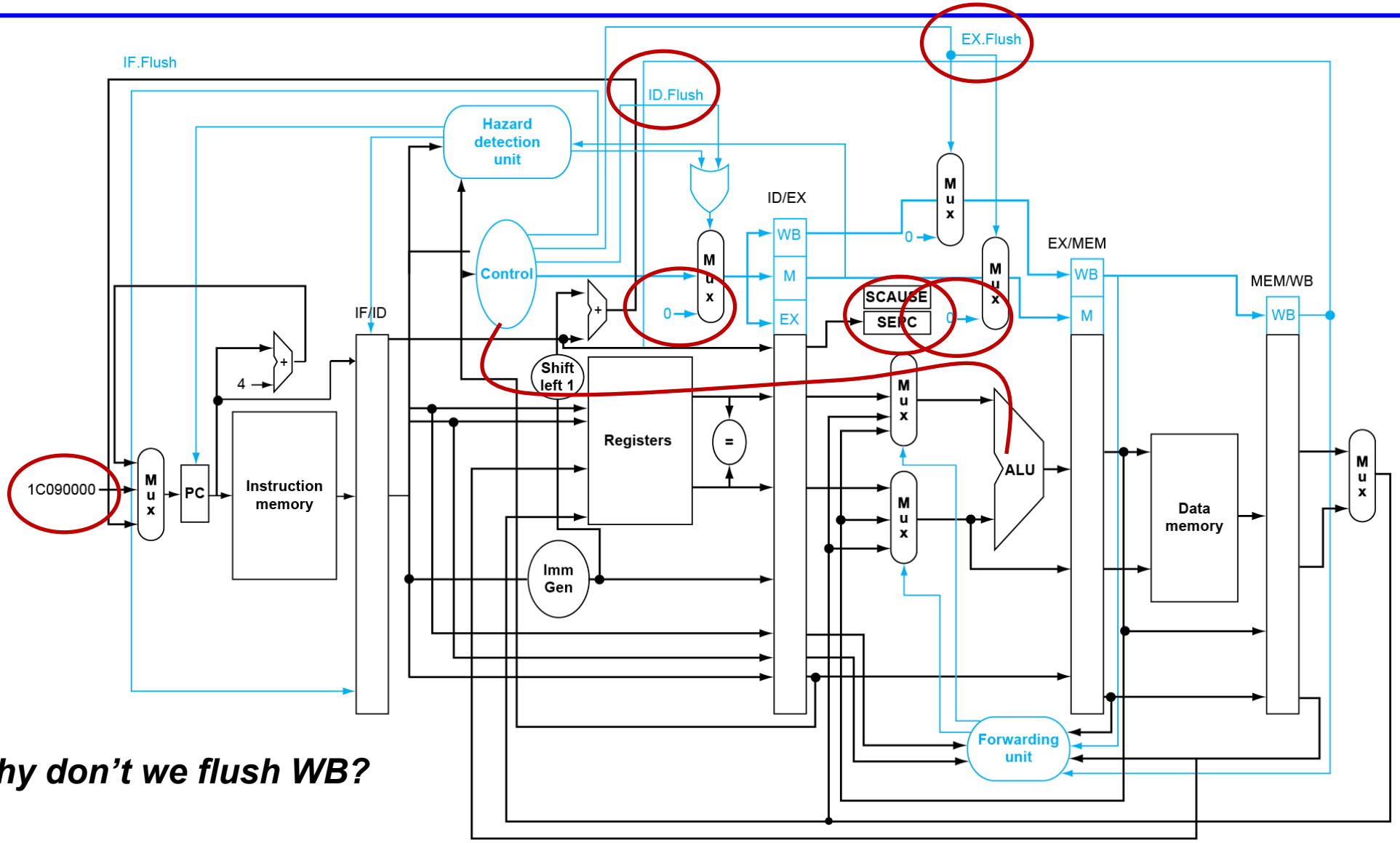
Note: another form of control hazard, similar to mispredicted branch

➔ *can reuse a lot of that hardware…*

Handler Actions

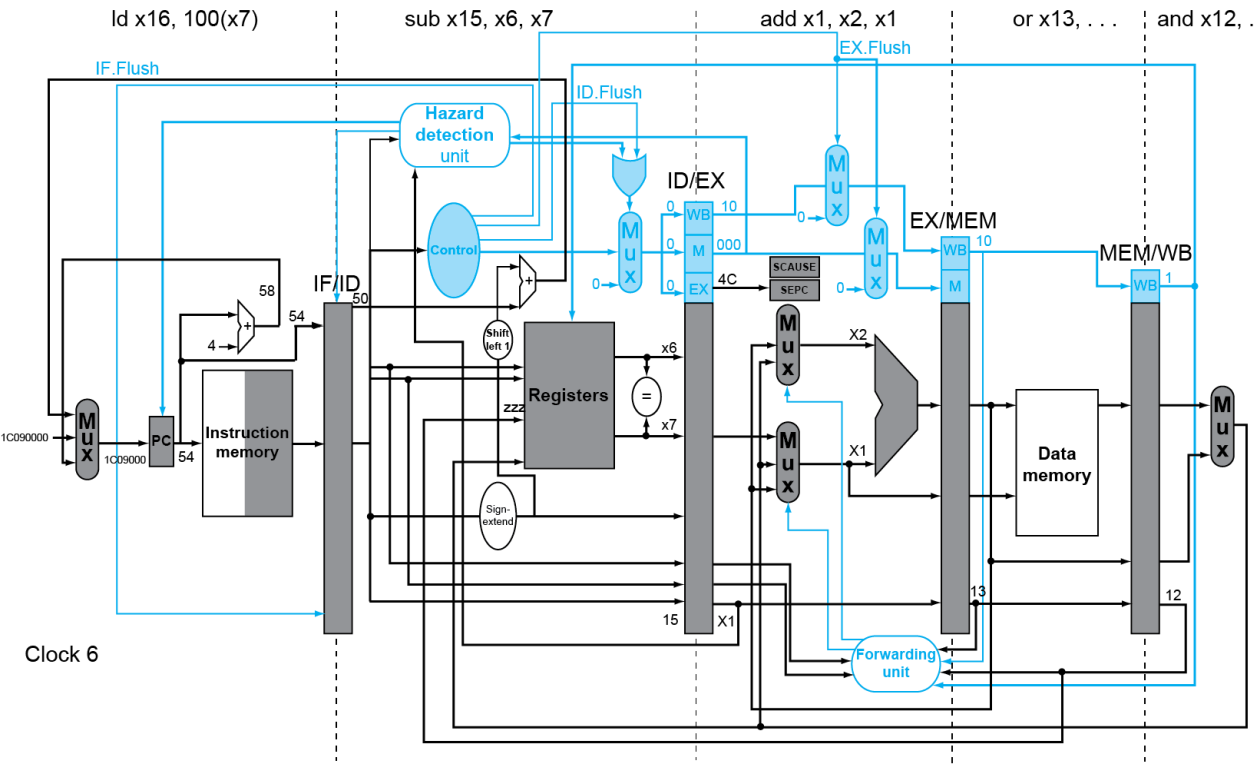
- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use SEPC to return to program
- Otherwise
  - Terminate program
  - Report error using SEPC, SCAUSE, …

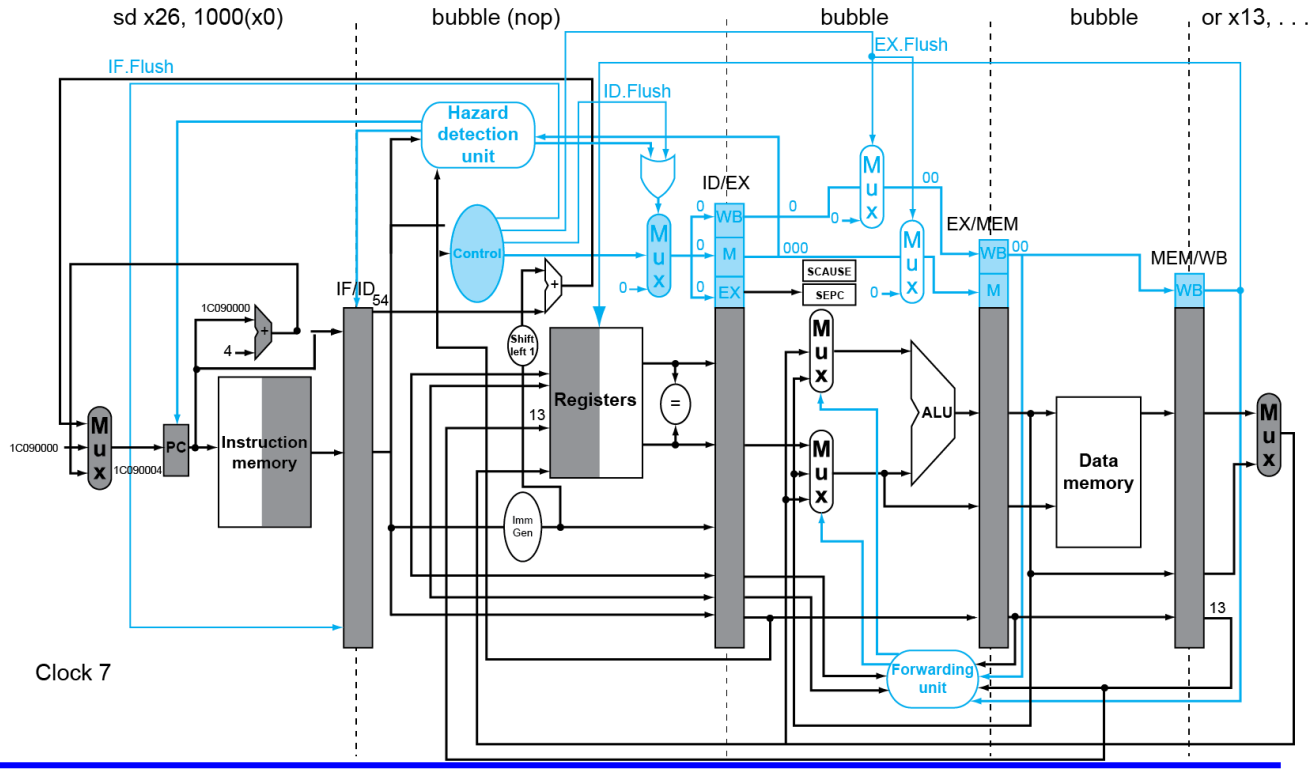
Pipeline with ALU exception handling



Q: Why don't we flush WB?

Exception example (ALU exception)





Concluding Remarks

- Pipelining boosts the processor's clock speed, but hazards can reduce performance again.

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

  - Unpipelined CPU time = X * 1 * 1
  - Unpipelined CPU time = X * CPI_{pipe} * 1/$N_{pipe\text{-}stages}$
    - CPI_{pipe} =

      Ideal pipeline CPI (=1 for single issue)

      + Data hazard stalls +

      + Control stalls

➔ Goal is to maximize CPI (at low clock cycle time)!

- Using many tricks to reduce data hazard stalls and control stalls:
  - Static / dynamic

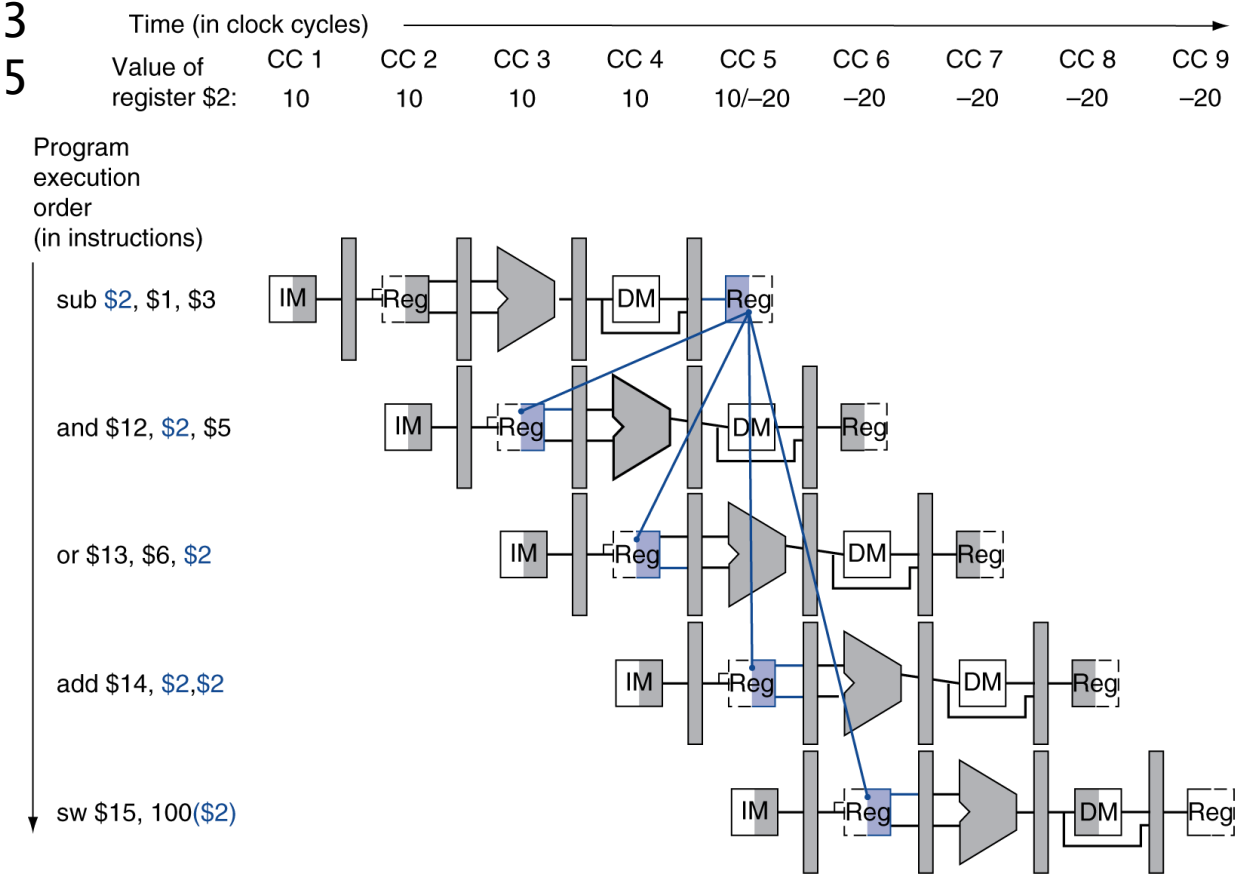
Representative exercises

Recapitulate...
Exercise 1: How many cycles lost if...

1. Compute–use data hazard, no forwarding?

E.g.

```
sub x2,  x1, x3
and x12, x2, x5
```

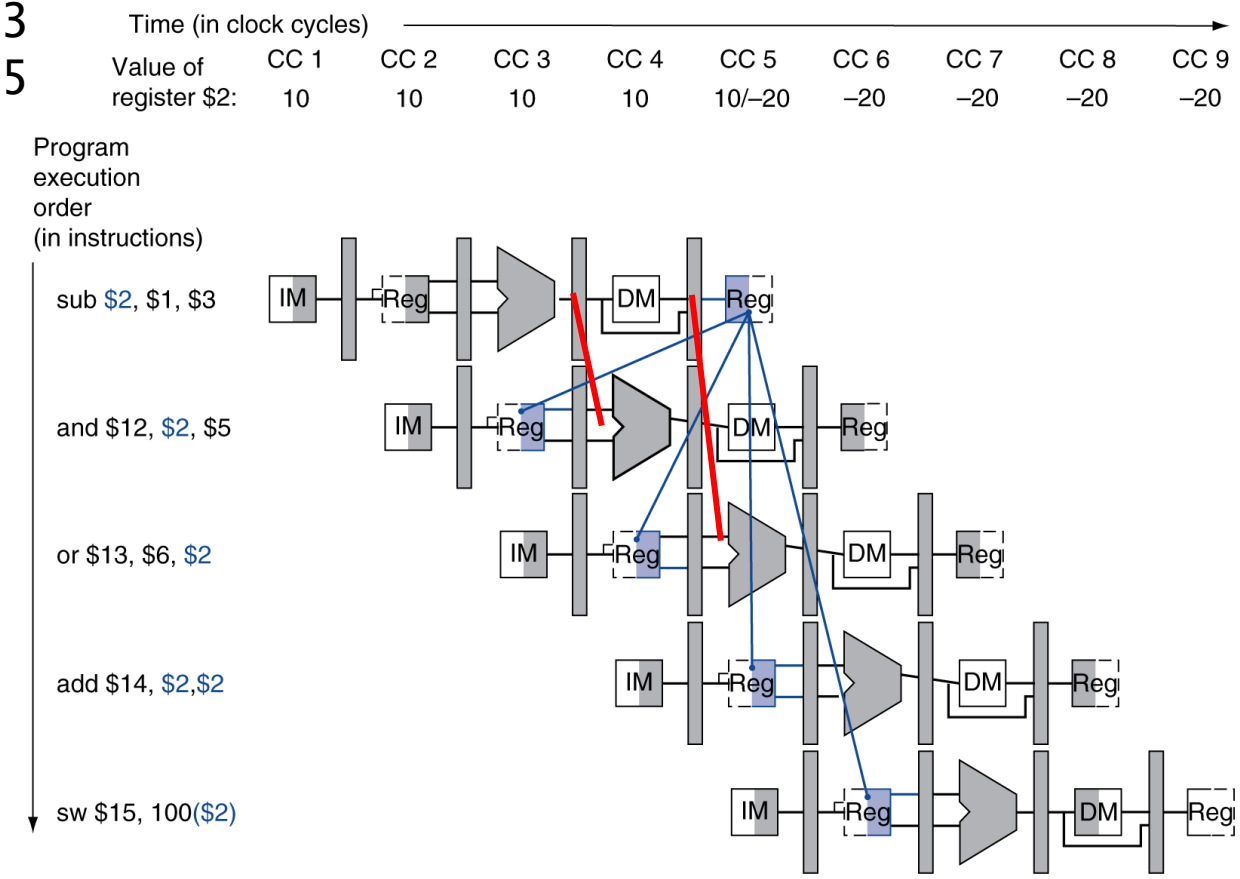


Recapitulate...
Exercise 1: How many cycles lost if...

2. Compute–use data hazard, with forwarding?

E.g.

```
sub x2,  x1, x3
and x12, x2, x5
```

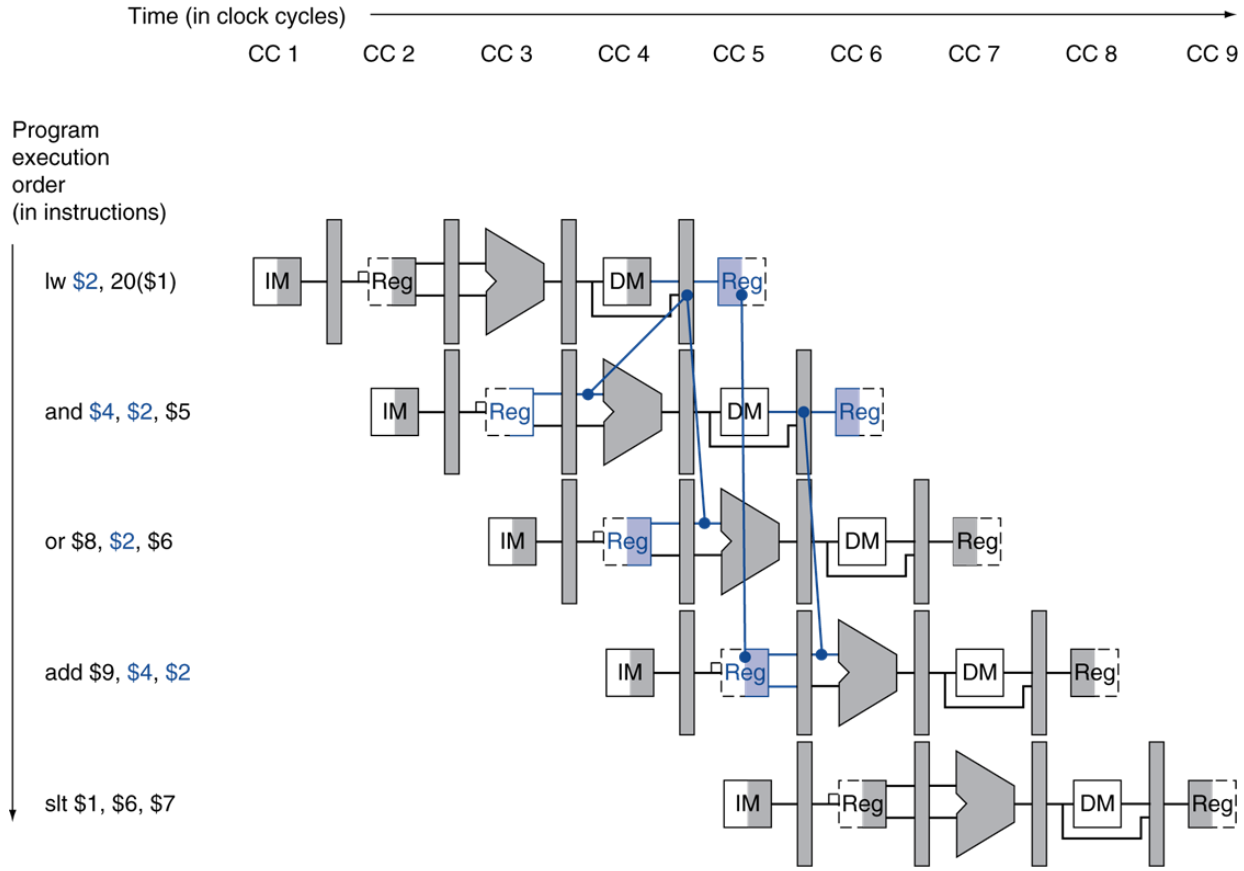


Recapitulate...
Exercise 1: How many cycles lost if...

3. Load–use data hazard, no forwarding?

E.g.

```
lw  x2, 20(x1)
and x4, x2, x5
```

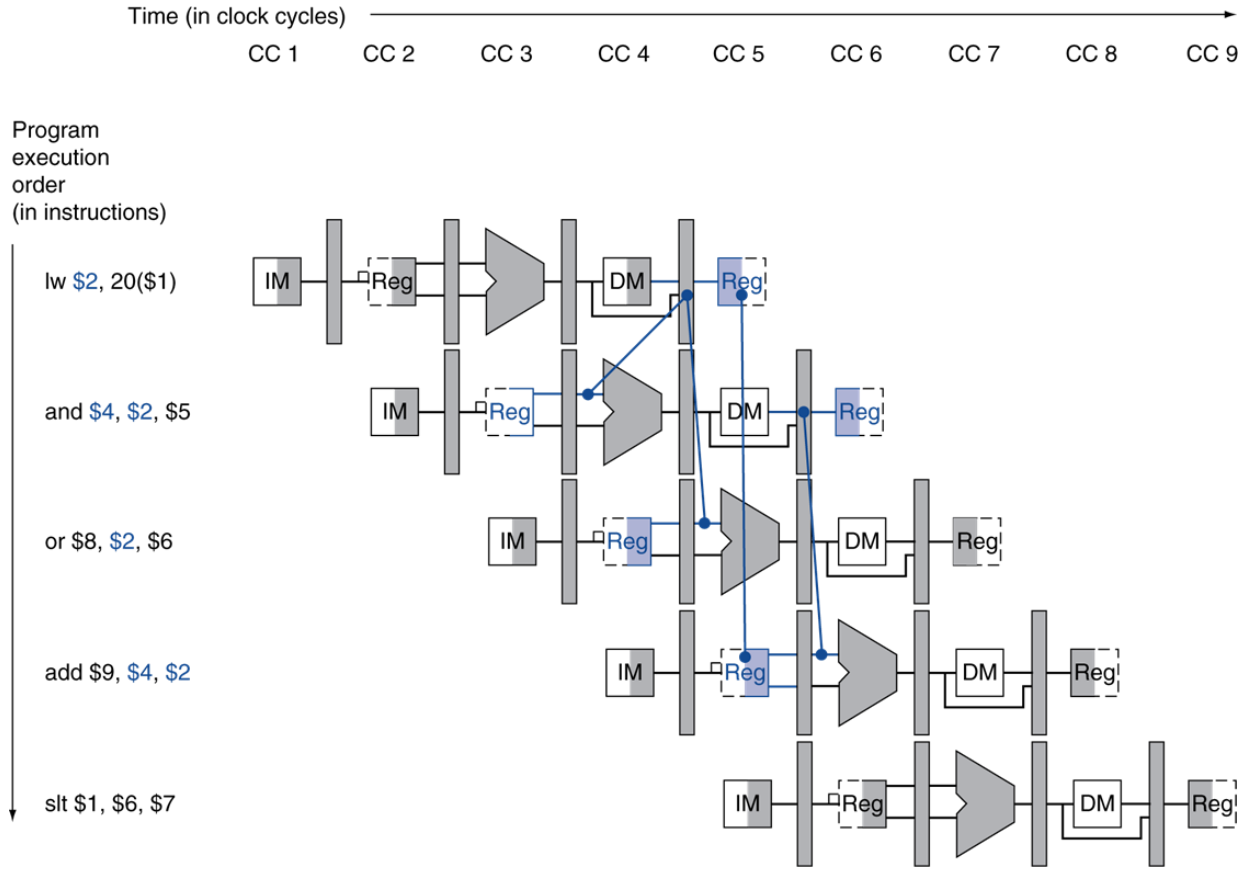


Recapitulate...
Exercise 1: How many cycles lost if...

4. Load–use data hazard, with forwarding?

E.g.

```
lw  x2, 20(x1)
and x4, x2, x5
```

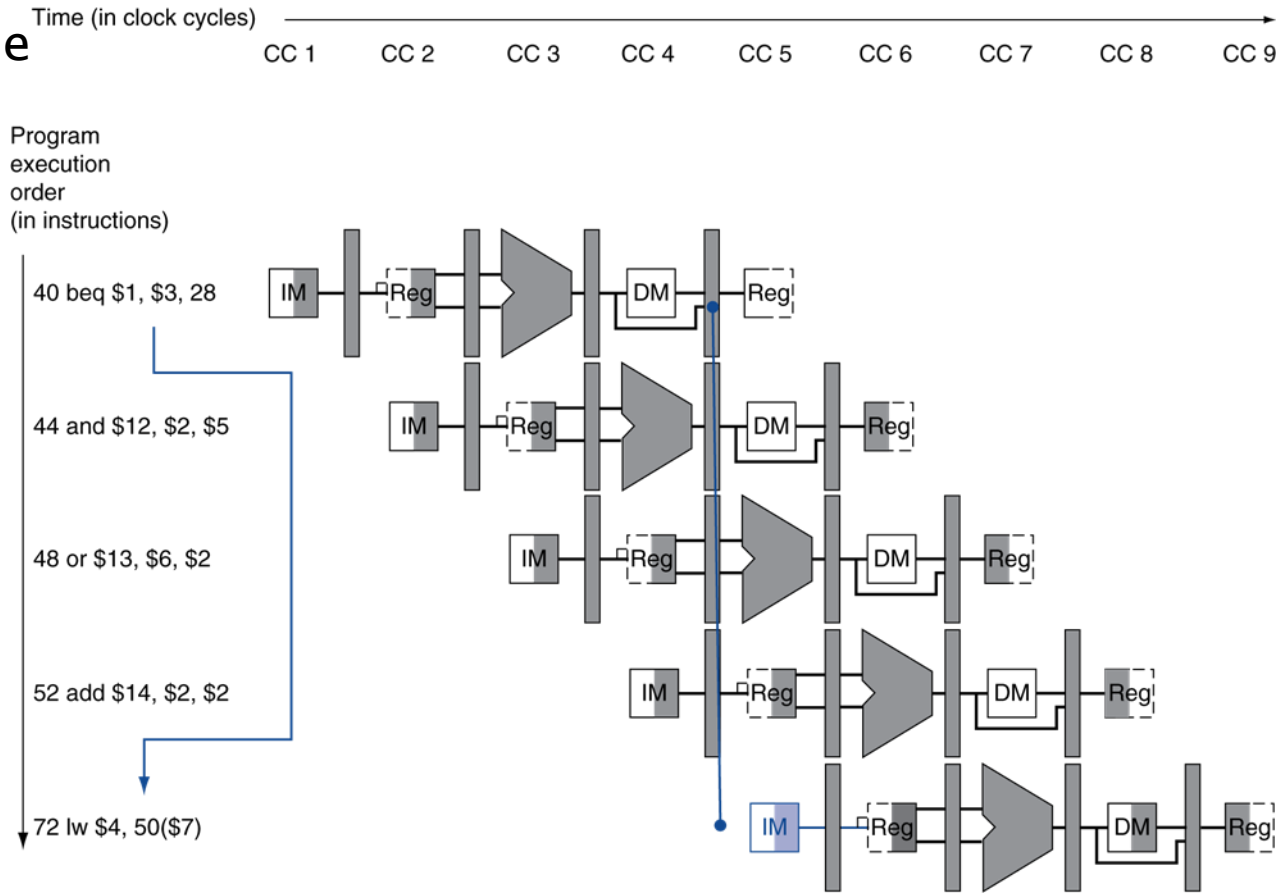


Recapitulate...
Exercise 1: How many cycles lost if...

5. Control hazard, eq detection in MEM stage?

E.g.

```
beq x1, x3 Else
…
```

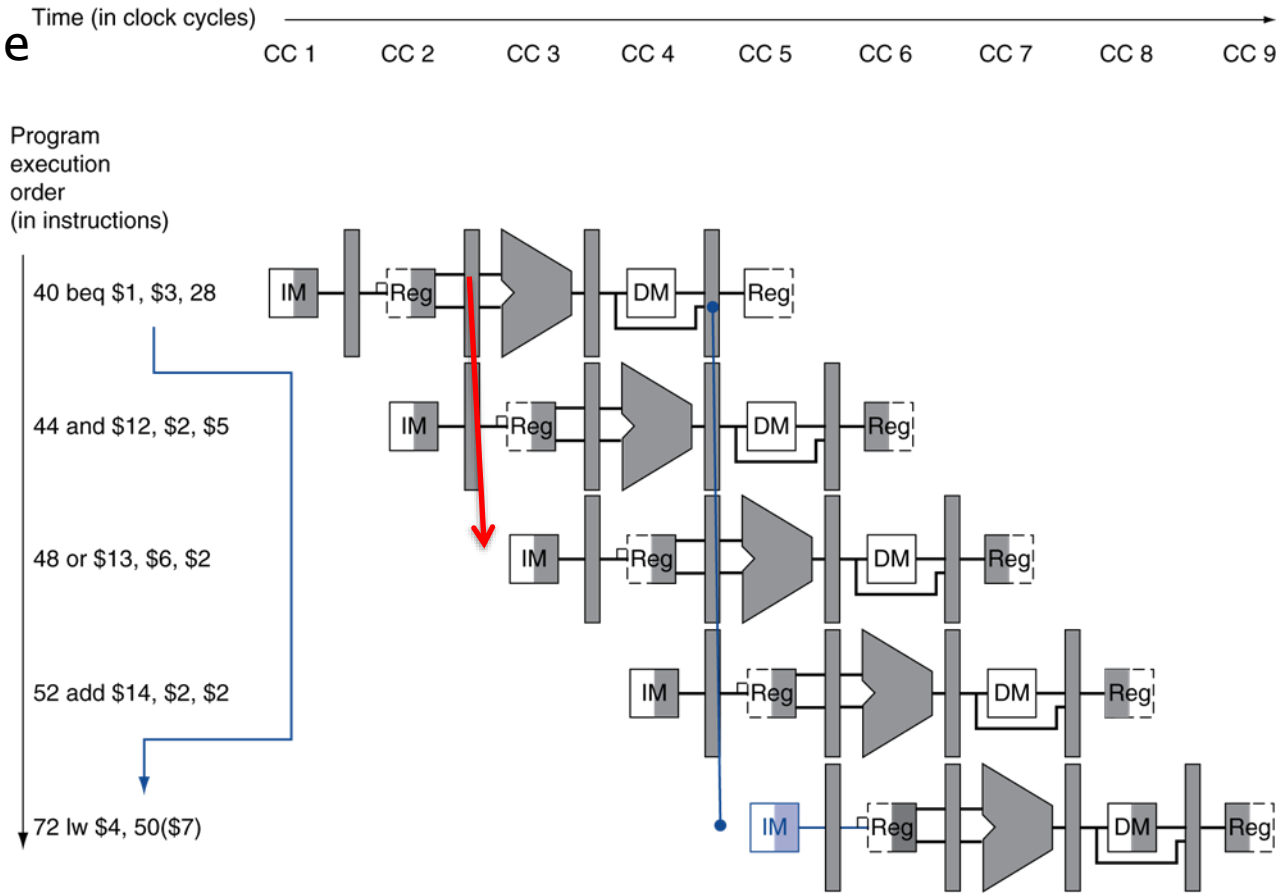


Recapitulate...
Exercise 1: How many cycles lost if...

6. Control hazard, eq detection in ID stage?

E.g.

```
beq x1, x3 Else
…
```

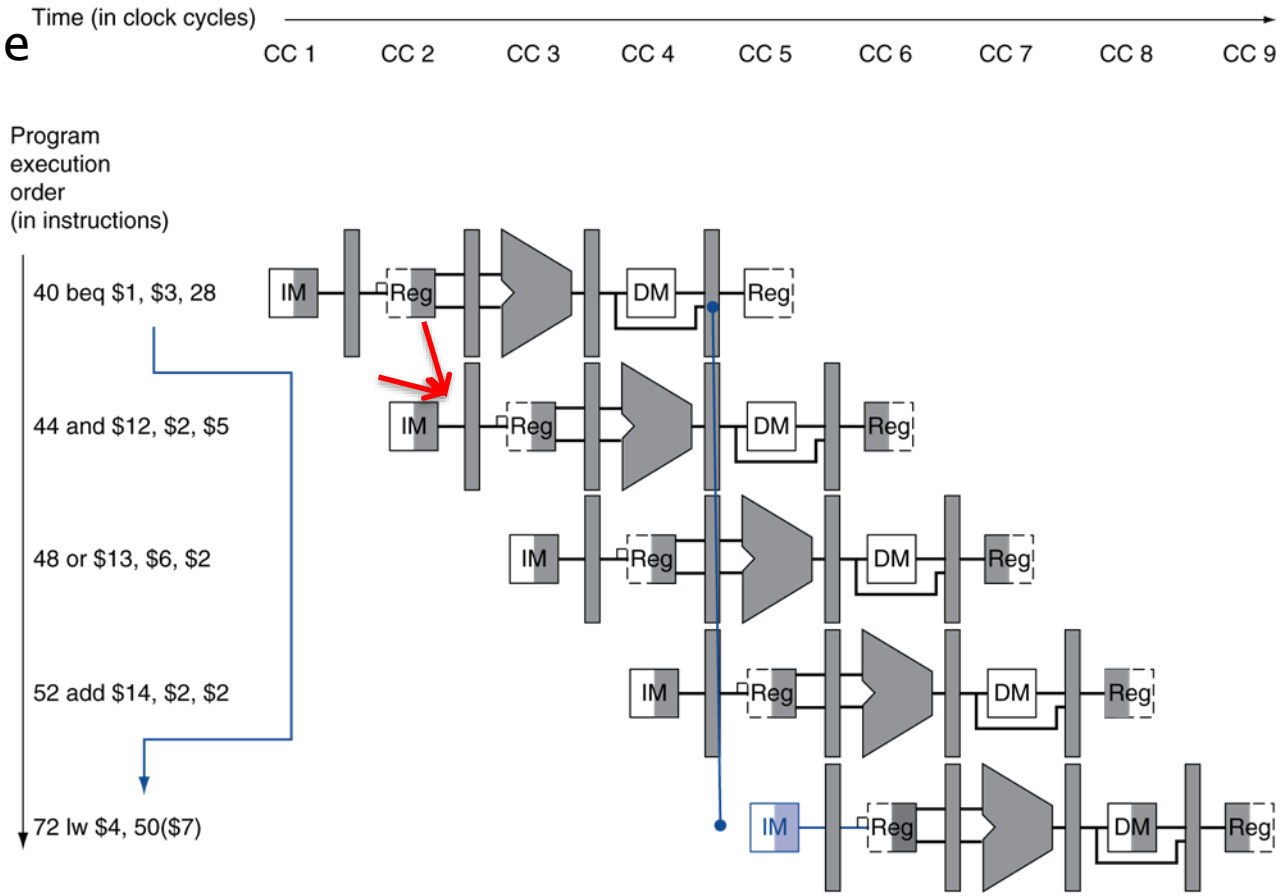


Recapitulate...
Exercise 1: How many cycles lost if...

6. Control hazard, eq detection in ID stage, BTH & BTB?

E.g.

```
beq x1, x3 Else
…
```

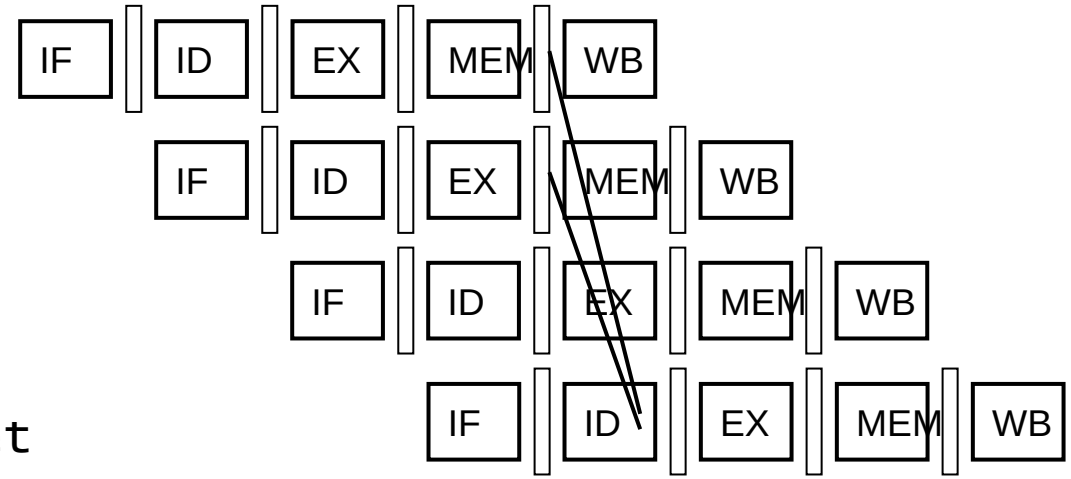


Recapitulate...
Exercise 1: How many cycles lost if...

7. Data hazard for branch, eq detection in ID stage, with forwarding?

E.g.

```
add x1, x2, x3
add x4, x5, x6
beq x1, x4, target
```

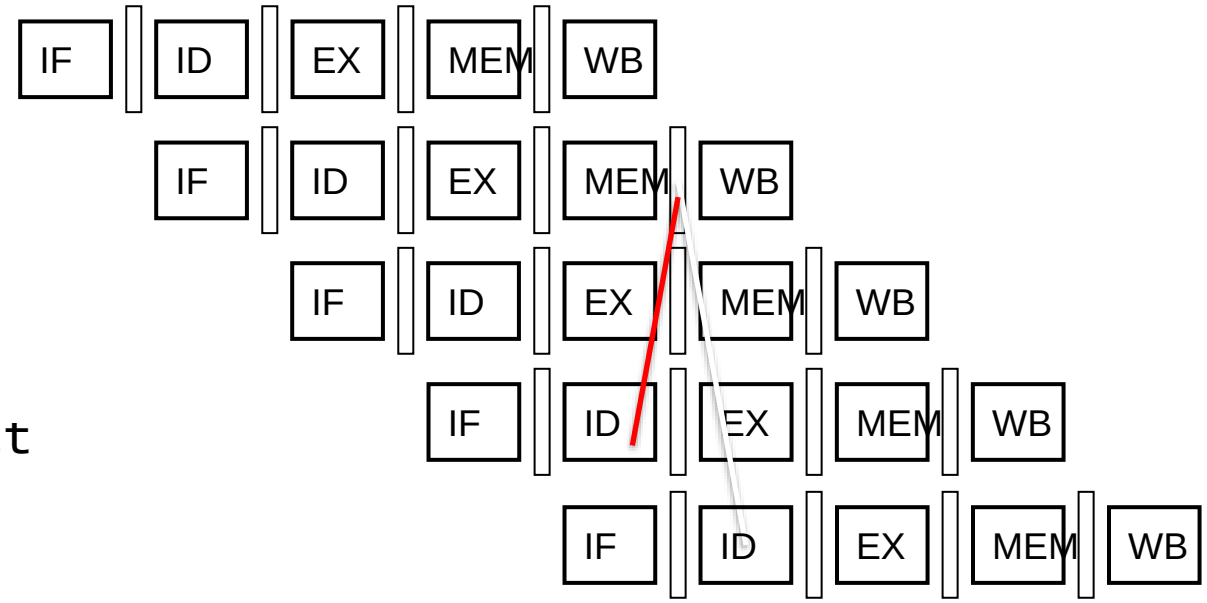


Recapitulate...
Exercise 1: How many cycles lost if...

8. Data (load-use) hazard for branch, eq detection in ID stage, with forwarding?

E.g.

```
add x1, x2, x3
lw x4, 20(x5)
beq x1, x4, target
```



Exercise 1: Solution

1. 2 cycles
2. 0 cycles
3. 2 cycles
4. 1 cycle
5. 3 cycles
6. 1 cycle
6. 0 cycles, if prediction correct...
7. 1 cycle; if using forwarding (otherwise 2)
8. 2 cycles; if using forwarding (otherwise also 2...) ➔ better if ID in EX stage...

Exercise2: Data hazards

```
sw x16, 12(x6)
lw x16, 8(x6)
add x5, x16, x4
sub x4, x16, x2
add x5, x16, x4
sw  x5, 8(x6)
```

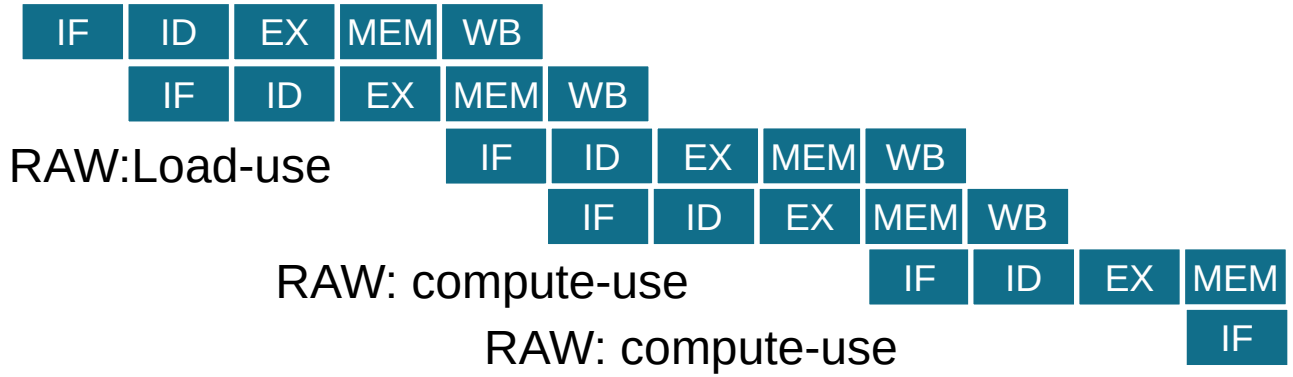
- How many cycles on regular 5-stage processor without bypassing?
- How many cycles on regular 5-stage processor with bypassing?

Extra: idem for

```
lw      $2, 20($1)
and     $4, $2,$5
or      $4, $4,$2
add     $9, $4,$2
```

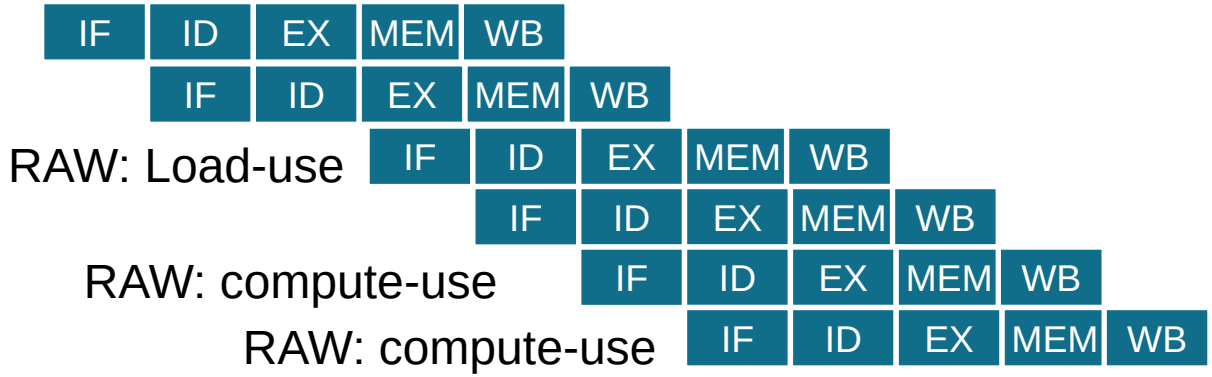
Exercise2: Solution

```
sw x16, 12(x6)
lw x16, 8(x6)
add x5, x16, x4
sub x4, x16, x2
add x5, x16, x4
sw  x5, 8(x6)
```



➔ 16 cycles

```
sw x16, 12(x6)
lw x16, 8(x6)
add x5, x16, x4
sub x4, x16, x2
add x5, x16, x4
sw  x5, 8(x6)
```



➔ 11 cycles

Exercise 3: Store hazards?

What happens if store instructions are dependent on other instructions?
E.g.

```
Compute-store:
add x16, x5, x4
sw x16, 12(x6)
```

```
Memory copy:
lw x16, 8(x6)
sw x16, 12(x6)
```

Can we modify the pipeline to mitigate this impact?

Exercise 3: solution

P371 (H&Pv4):

Forwarding can also help with hazards when store instructions are dependent on other instructions. Since they use just one data value during the MEM stage, forwarding is easy. However, consider loads immediately followed by stores, useful when performing memory-to-memory copies in the MIPS architecture. Since copies are frequent, we need to add more forwarding hardware to make them run faster. If we were to redraw Figure 4.53, replacing the sub and instructions with lw and sw, we would see that it is possible to avoid a stall, since the data exists in the MEM/WB register of a load instruction in time for its use in the MEM stage of a store instruction. We would need to add forwarding into the memory access stage for this option.

FW from MEM to EX stage not OK. It solves the comp-store hazard, but not the load-store… ➔ exam!!

Exercise4: Static branch handling

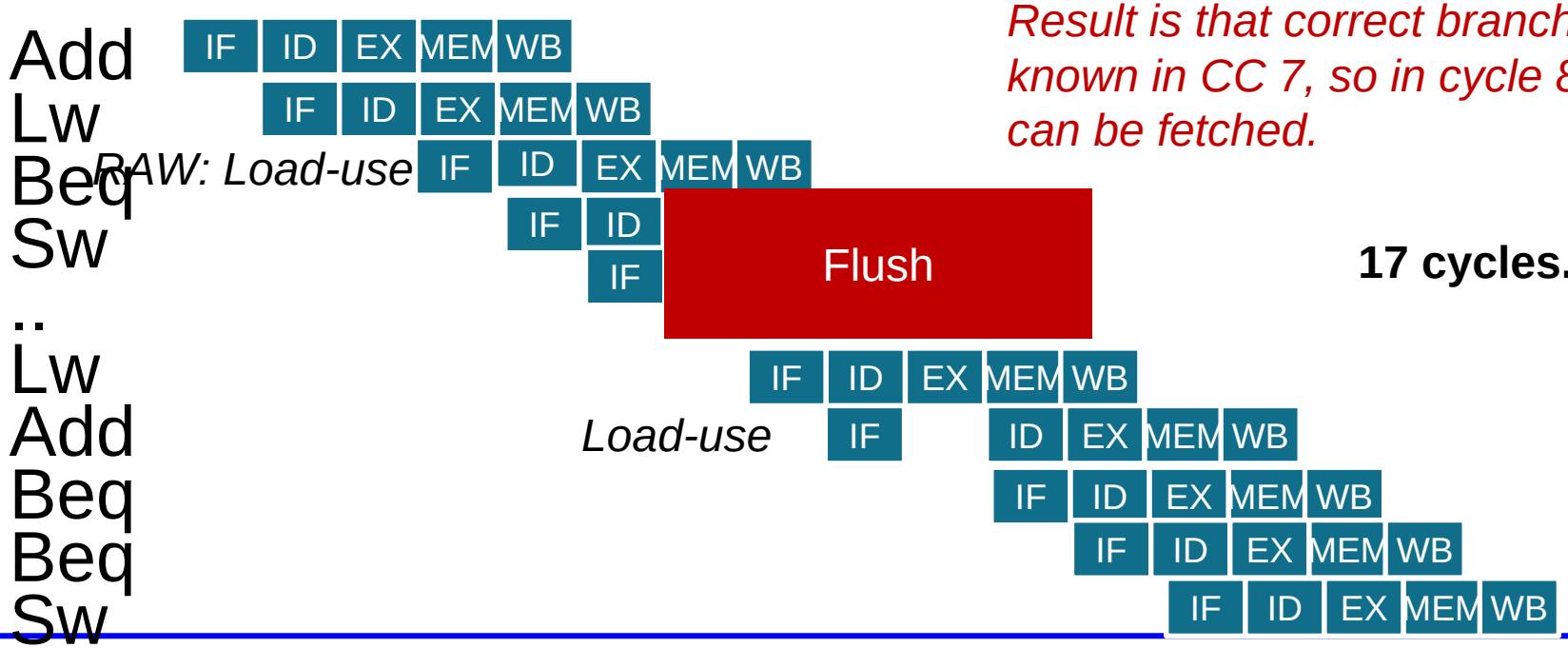
```
        add R3, R3, R4
        lw  R2, 0(R1)
Label1: beq R2, R0, Label2    (Not taken once, then taken)
        lw R3, 0(R2)
        add R1, R3, R1
        beq R3, R0, Label1    (Taken)
Label2: sw R1, 0(R2)
```

- Assume: *5-stage pipeline, full fowarding, simple predict-taken branch handling, branch execution in EX-stage*
  - How many cycles to execute code?
  - Would branch execution in the ID-stage help? How much?
  - What additional logic would be needed to support all this?
  - Would more advanced predictors help (2-bit)?

Exercise4: Solution

```
        add R3, R3, R4
        lw  R2, 0(R1)
Label1: beq R2, R0, Label2     (Not taken once, then taken)
        lw R3, 0(R2)
        add R1, R3, R1
        beq R3, R0, Label1     (Taken)
Label2: sw R1, 0(R2)
```

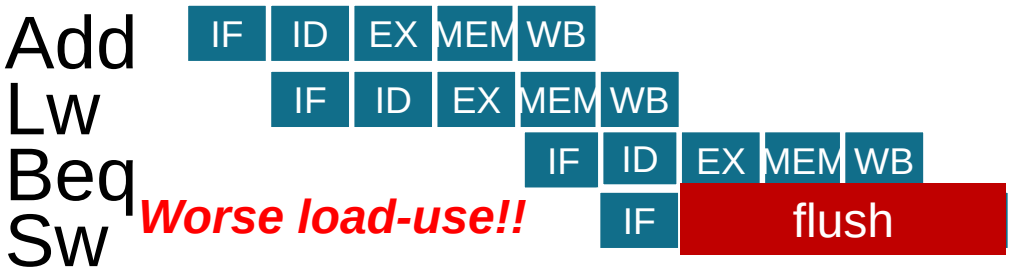
Be careful: data hazard from instr 2 to 3 causing 1 cycle delay (from MEM ot EX stage)
Result is that correct branching address is known in CC 7, so in cycle 8 correct instruction can be fetched.



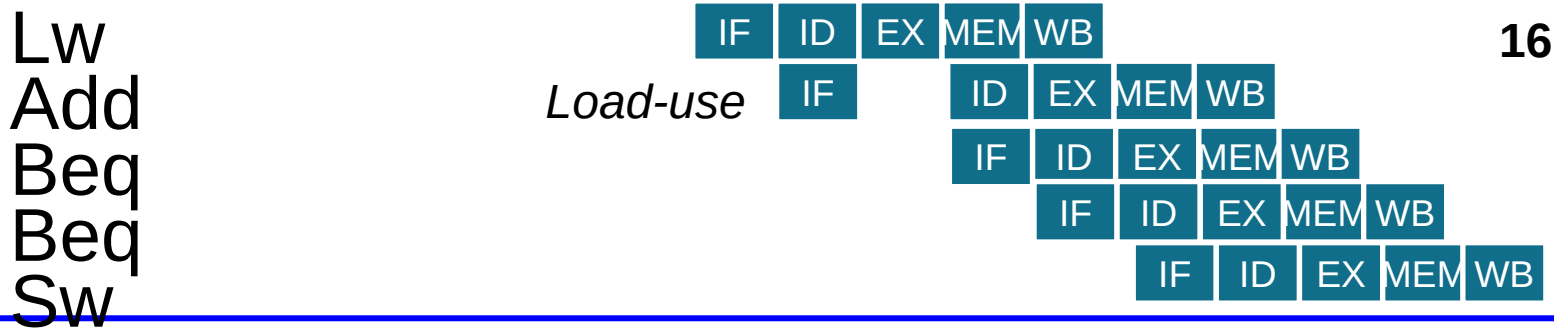
17 cycles...

Exercise4: Solution

```
        add R3, R3, R4
        lw  R2, 0(R1)
Label1: beq R2, R0, Label2     (Not taken once, then taken)
        lw R3, 0(R2)
        add R1, R3, R1
        beq R3, R0, Label1     (Taken)
Label2: sw R1, 0(R2)
```



Be careful: data hazard from instr 2 to 3 causing now 2 cycles delay (from MEM ot ID stage), because the equanlity detection is now done in the ID stage!!!
Result is that correct branching address is known in CC 6, so in cycle 7 correct instruction can be fetched.



16 cycles...

Exercise4: Solution

- The **hazard detection logic** must detect situations when the branch depends on the result of **the previous R-type instruction, or on the result of two previous loads**. When the branch uses the values of its register operands in its ID stage, the R-type instruction's result is still being generated in the EX stage. Thus we must stall the processor and repeat the ID stage of the branch in the next cycle. Similarly, if the branch depends on a load that immediately precedes it, the result of the load is only generated two cycles after the branch enters the ID stage, so we must stall the branch for two cycles. Finally, if the branch depends on a load that is the second-previous instruction, the load is completing its MEM stage when the branch is in its ID stage, so we must stall the branch for one cycle. In all three cases, **the hazard is a data hazard**. Note that in all the R-type instruction case we assume that the values of preceding instructions are **forwarded to the ID stage of the branch if possible**. ➔ draw this!!!

Exercise5: Dynamic branch handling

```
        add R3, R3, R4
        lw  R2, 0(R1)
Loop:   lw R3, 0(R2)
        beq R3, R0, Else     (T, T, NT, T, T, NT, T, T, NT,…)
        add R1, R3, R1
Else: sw R1, 0(R2)
        add R2, R4, R3
        beq R2, R0, Loop    (9xT, 1xNT, 9xT, 1xNT,…)
```

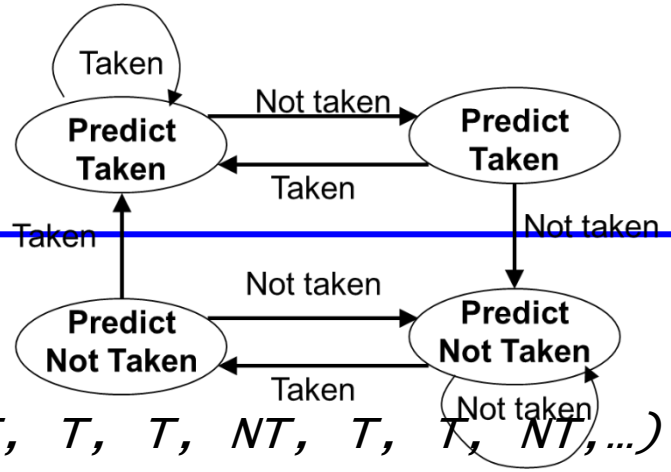
- How many cycles wasted per complete passing due to mispredictions in regular 5-stage pipeline, branch execution in EX-stage, when using:
  - Simple prediction (predict not-taken)
  - Static prediction
  - 2-bit dynamic prediction (n=2, m=0)
- What would you propose to further optimize the execution of this code? (SW or HW modifications)

Exercise5: Solution

```
        add R3, R3, R4
        lw  R2, 0(R1)
Loop:   lw R3, 0(R2)
        beq R3, R0, Else     (T, T, NT, T, T, NT, T, T, NT,…)
        add R1, R3, R1
Else: sw R1, 0(R2)
        add R2, R4, R3
        beq R2, R0, Loop    (9xT, 1xNT, 9xT, 1xNT,…)
```

- Each branch that is not correctly predicted will cause 3 stall cycles
- Simple prediction (pred. not-taken):
  - 1-st branch 66% wrong: 0.66*3 cycles; 2nd brach: 90% wrong= 0.9*3 cycles ➔ 4.68 cycles
- Static prediction (FW: not-taken; BW: taken)
  - 1-st branch 66% wrong: 0.66*3 cycles; 2nd brach: 10% wrong= 0.1*3 cycles ➔ 2.28 cycles

Exercise5: Solution




```
          add R3, R3, R4
          lw  R2, 0(R1)
Loop:     lw R3, 0(R2)
          beq R3, R0, Else     (T, T, NT, T, T, NT, T, T, NT,…)
          add R1, R3, R1
Else: sw R1, 0(R2)
          add R2, R4, R3
          beq R2, R0, Loop     (9xT, 1xNT, 9xT, 1xNT,…)
```

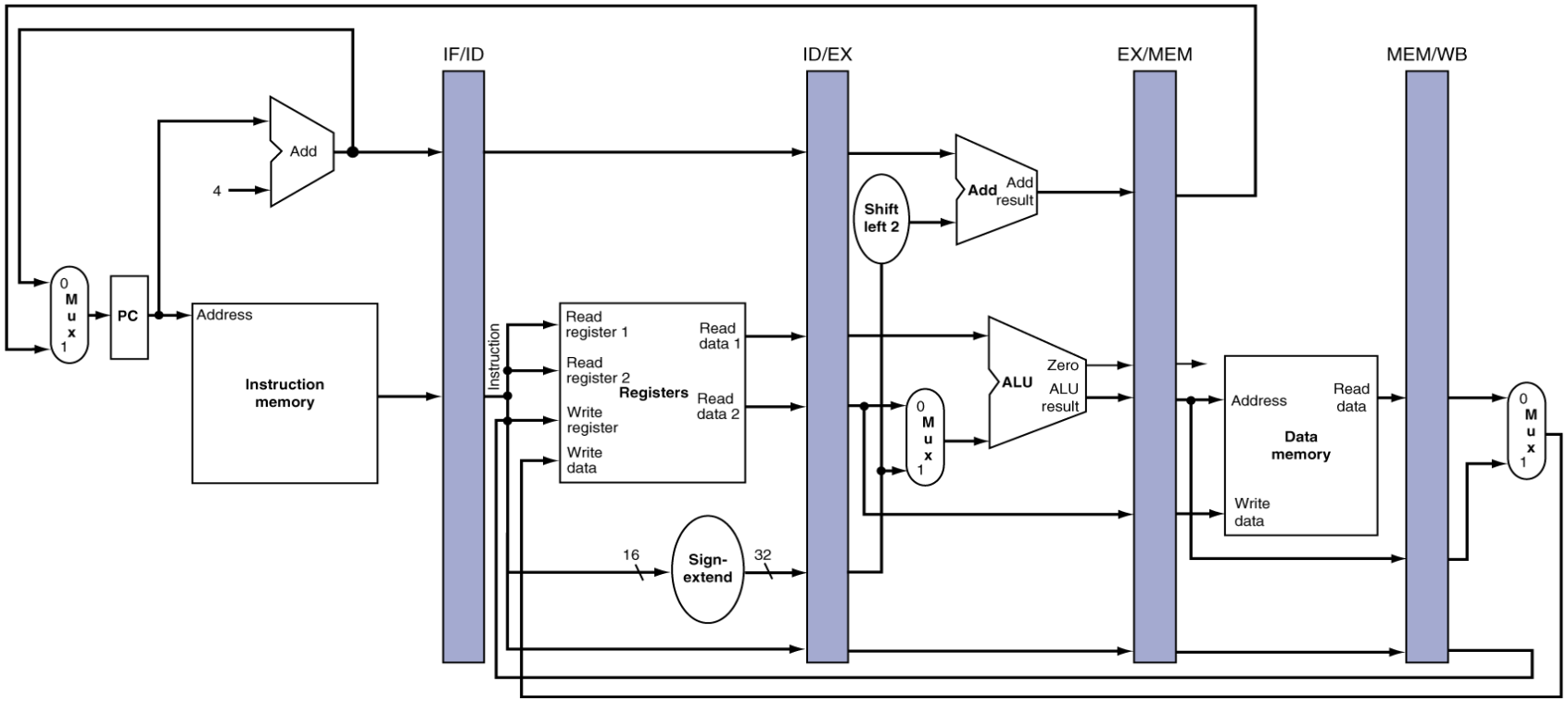
- Dynamic prediction (2-bit)
  - 1-st branch 33% wrong: 0.33*3 cycles; 2nd brach: 10% wrong= 0.1*3 cycles ➔ 1.3 cycles
- Modifications:
  - Branch comput to ID stage (but needs extra HW and bypassing / hazard detection logic! (Possibly more data hazards....))

Exercise6: Modify the pipeline

`addM Rd, Rs1, Rs2` *Rd = Rs2 + Mem[Rs1]*

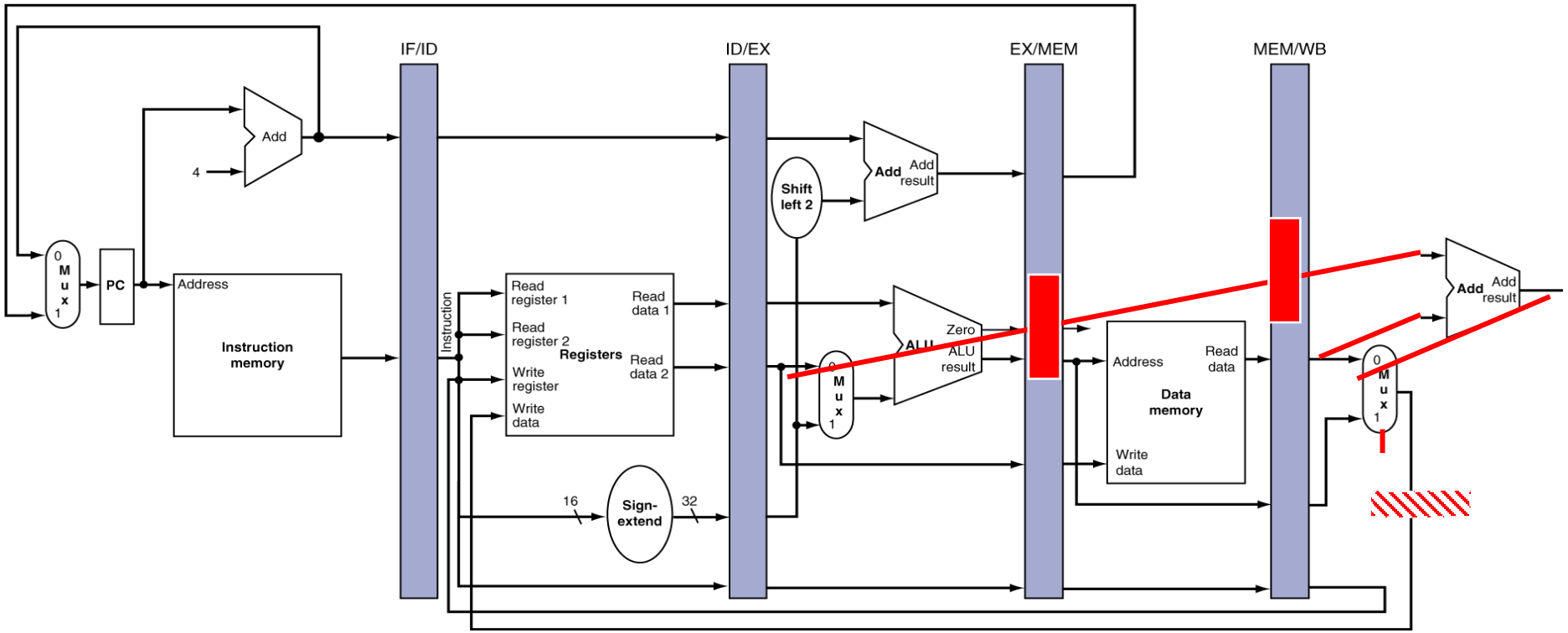
Add up the content of the register Rs2 with the content in the data memory at address given by register Rs1, store the result in Rd.

- What must be changed in the pipeline to add this instruction to the SC-V ISA?
- Does this have impact on the bypassing scheme


IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

Exercise6: Solution

- This instruction behaves like a normal load until the end of the MEM stage. After that, it behaves like an ADD, so we need another stage after MEM to compute the result, and we need additional wiring to get the value of Rt to this stage.
- We need to add a control signal that selects what the new stage does (just pass the value from memory through, or add the register value to it).
- can be used when trying to compute a sum of array elements.


IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
16
Sign-extend
32
Address
Data memory
Read data
Write data