



# ***Pipelined Architecture with solutions to data & control hazards***

# Pipeline Processing Hazards

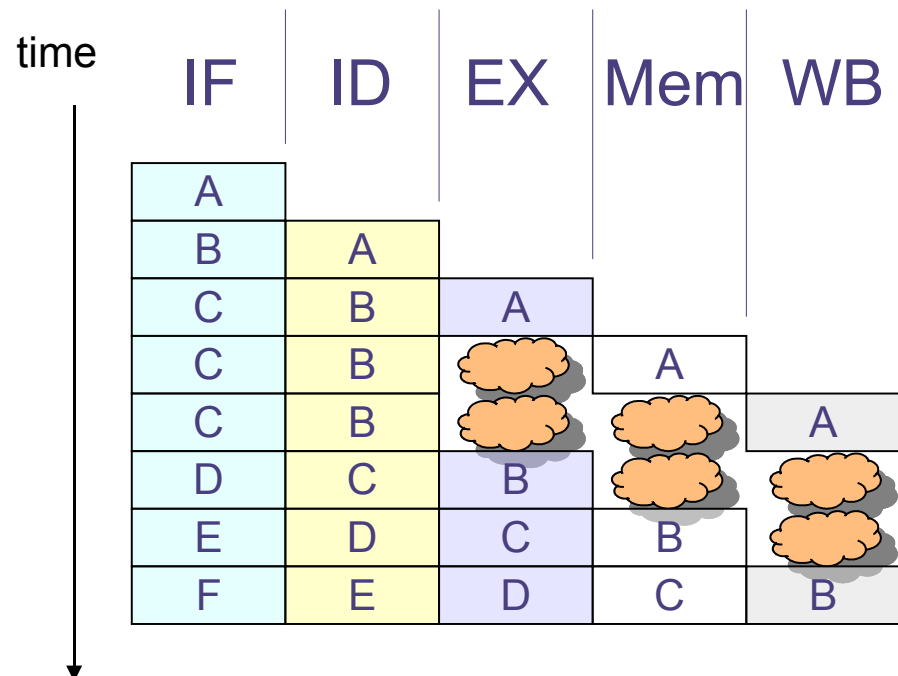
---

- Structural Hazard
  - hardware duplication
- Data Hazard
  - Pipeline Stall
  - Software (machine code) optimization
  - Forwarding
- Control Hazard
  - Pipeline Flush (Instruction Invalidation)
  - Delayed Branching
  - Early Branch Detection
  - Branch History Table

# Pipeline Stall

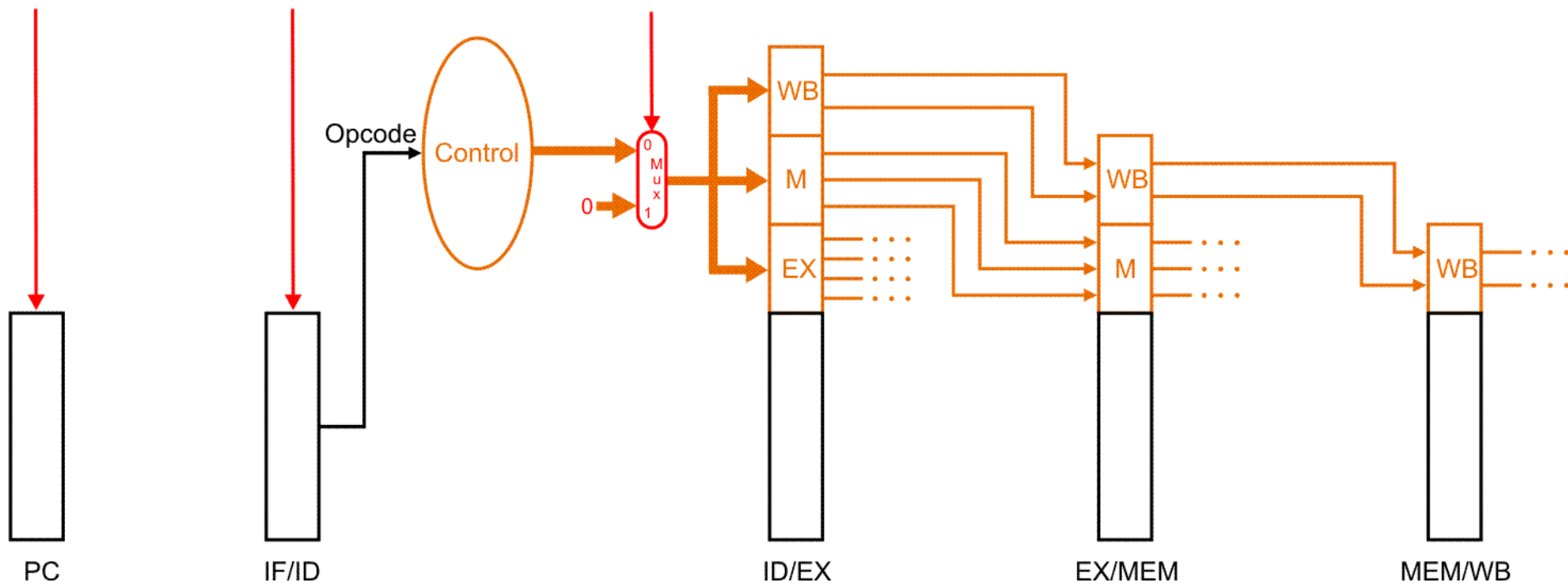
- Some stages must be repeated – other invalidated
- Reading the register being modified:
  - the same register can be referred to in ID (read) and in WB (write) stage – the writing can be done before (half clock cycle) reading

**A) ADD R1 , R2 , R3**  
**B) SUB R4 , R3 , R5**  
**C) MUL R4 , R3 , R1**  
**D) . . .**  
**E) . . .**  
**F) . . .**



# Hardware Pipeline Stall

- Deactivation ( $\rightarrow 0$ ) of control signals for stages: Ex, Mem and WB
- Postponed writing to PC and IF/ID



# Software "Pipeline Stall"

- Software correction of data flow with NOP (*No Operation*)
  - not truly optimization, but might be occasionally necessary when hardware mechanisms are insufficient

```
next: LW    R1, 0(R3)
      MUL   R1, R1, R1
      SW    R1, 0(R3)
      SUBI  R3, #4, R3
      BNE   R0, R3, next
      ...
```

all the data hazards  
"solved" with NOPs

```
next: LW    R1, 0(R3)
      NOP
      MUL   R1, R1, R1
      NOP
      NOP
      SW    R1, 0(R3)
      SUBI  R3, #4, R3
      NOP
      NOP
      BNE   R0, R3, next
      ...
```

# Software Optimization for Architecture

- Static: optimization at compilation time  
(*optimising compiler*)
  - e.g. gcc -O<sub>n</sub> -march=xxx
- Dynamic: at run-time: executing instructions in optimal order detected by hardware
  - dynamic scheduling
  - rename registers
  - out of order execution
  - speculative execution

Beyond the scope of this lecture





# GCC Settings

`gcc -o test test.c -O3 -march=athlon`

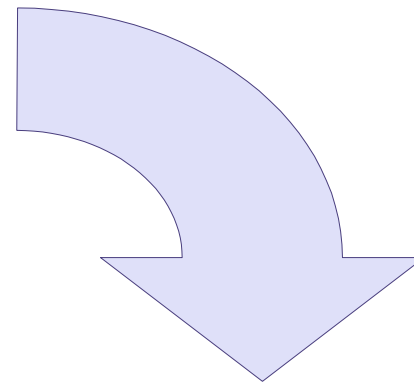
Target CPU Types	-march= Type
i386 DX/SX/CX/EX/SL	i386
i486 DX/SX/DX2/SL/SX2/DX4	i486
487	i486
Pentium	pentium
Pentium MMX	pentium-mmx
Pentium Pro	pentiumpro
Pentium II	pentium2
Celeron	pentium2
Pentium III	pentium3
Pentium 4	pentium4
Via C3	c3
Winchip 2	winchip2
Winchip C6-2	winchip-c6
AMD K5	i586
AMD K6	k6
AMD K6 II	k6-2
AMD K6 III	k6-3
AMD Athlon	athlon
AMD Athlon 4	athlon
AMD Athlon XP/MP	athlon
AMD Duron	athlon
AMD Tbird	athlon-tbird

Optimization	Included in Level			
	-O1	-O2	-Os	-O3
defer-pop	●	●	●	●
thread-jumps	●	●	●	●
branch-probabilities	●	●	●	●
cprop-registers	●	●	●	●
guess-branch-probability	●	●	●	●
omit-frame-pointer	●	●	●	●
align-loops	○	●	○	●
align-jumps	○	●	○	●
align-labels	○	●	○	●
align-functions	○	●	○	●
optimize-sibling-calls	○	●	●	●
cse-follow-jumps	○	●	●	●
cse-skip-blocks	○	●	●	●
gcse	○	●	●	●
expensive-optimizations	○	●	●	●
strength-reduce	○	●	●	●
rerun-cse-after-loop	○	●	●	●
rerun-loop-opt	○	●	●	●
caller-saves	○	●	●	●
force-mem	○	●	●	●
peephole2	○	●	●	●
regmove	○	●	●	●
strict-aliasing	○	●	●	●
delete-null-pointer-checks	○	●	●	●
reorder-blocks	○	●	●	●
schedule-insns	○	●	●	●
schedule-insns2	○	●	●	●
inline-functions	○	○	○	●
rename-registers	○	○	○	●

# Static Optimization Example

- For pipelined architecture with *Single Delay Slot*

```
next: LW    R1, 0(R3)
      MUL   R1, R1, R1
      SW    R1, 0(R3)
      SUBI   R3, #4, R3
      BNE   R0, R3, next
      ...
```

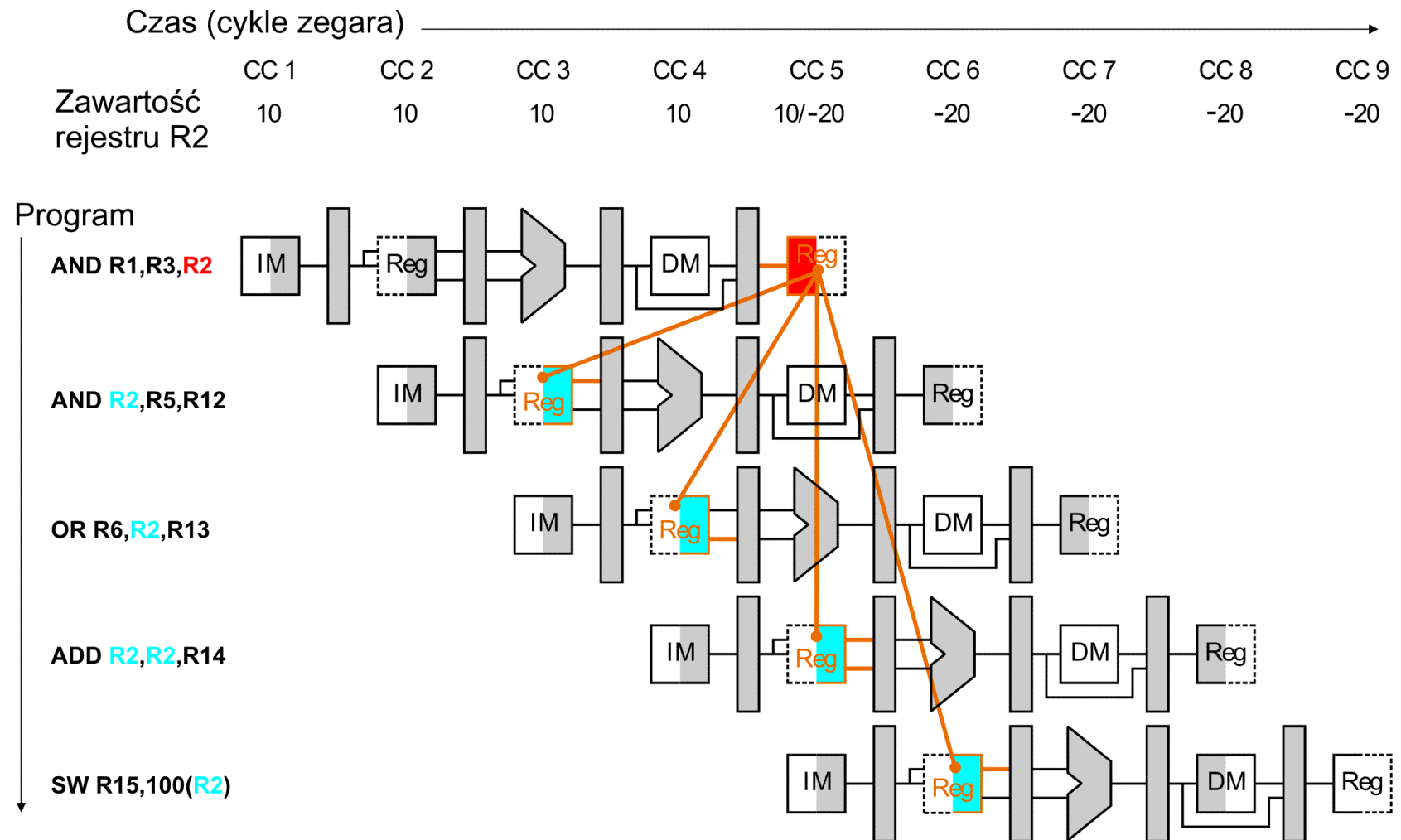


```
next: LW    R1, 0(R3)
      SUBI   R3, #4, R3
      MUL   R1, R1, R1
      BNE   R0, R3, next
      SW    R1, 4(R3)
      ...
```



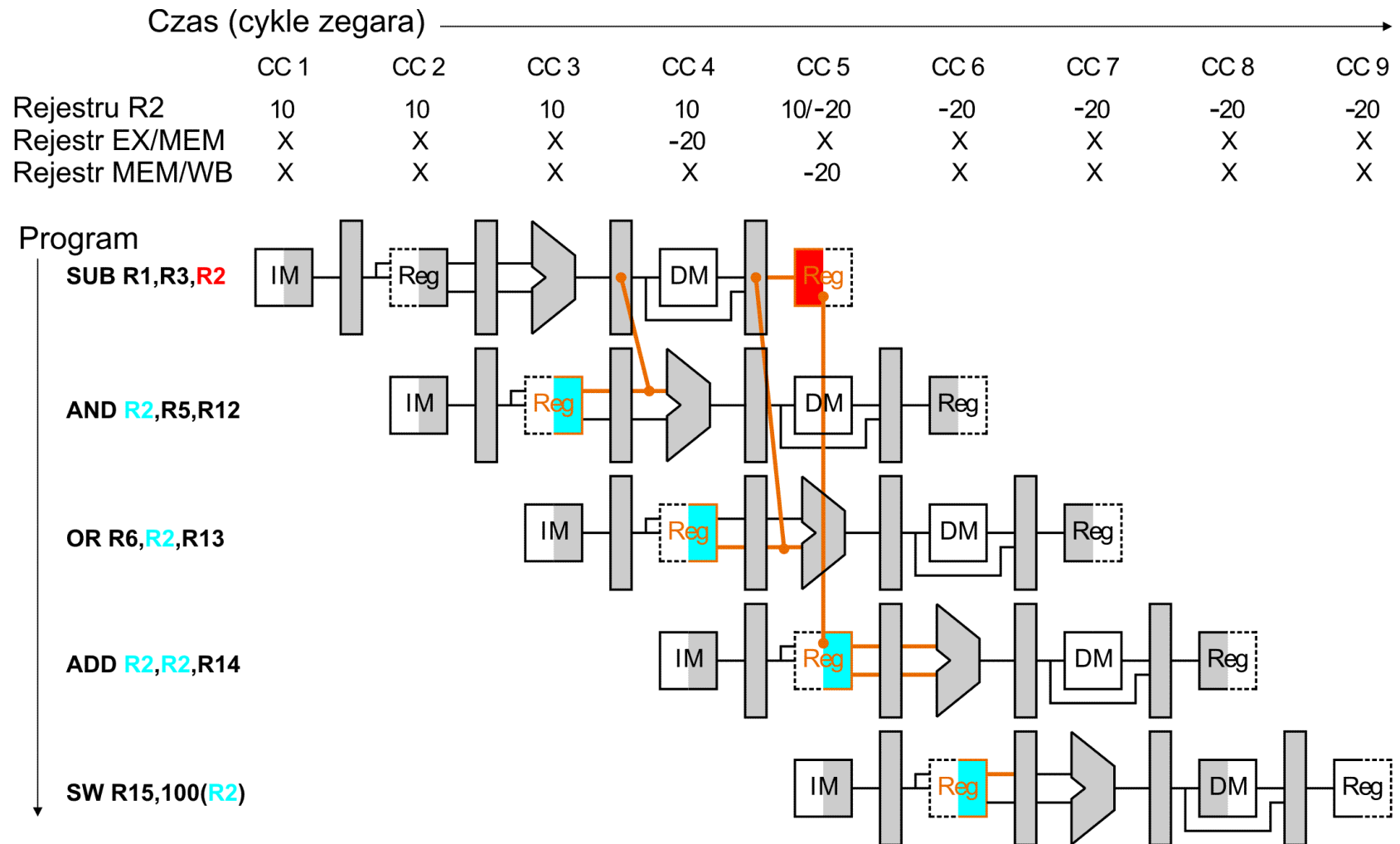
# Forwarding

## Efficient hardware solution to most data hazards



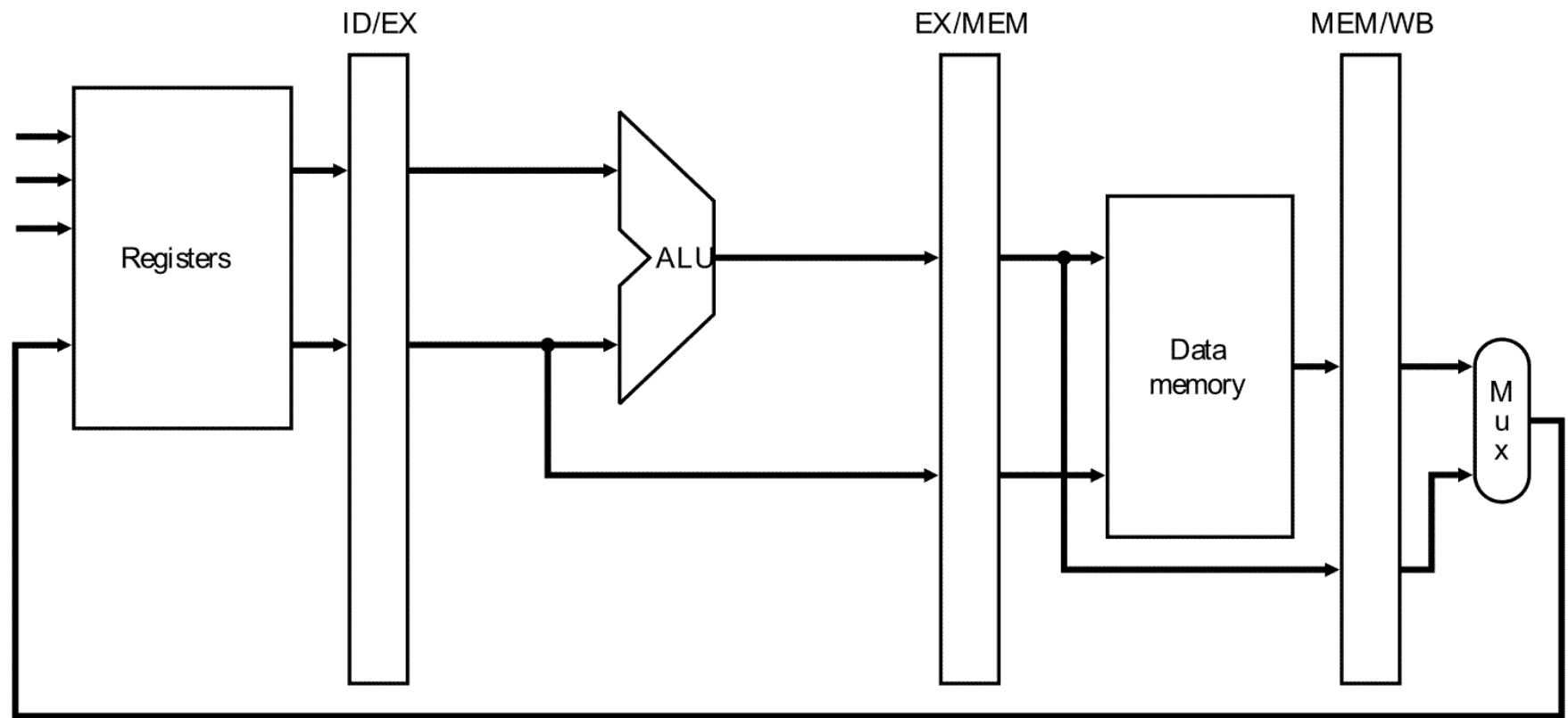
# Forwarding

- Idea: direct data access from intermediate registers



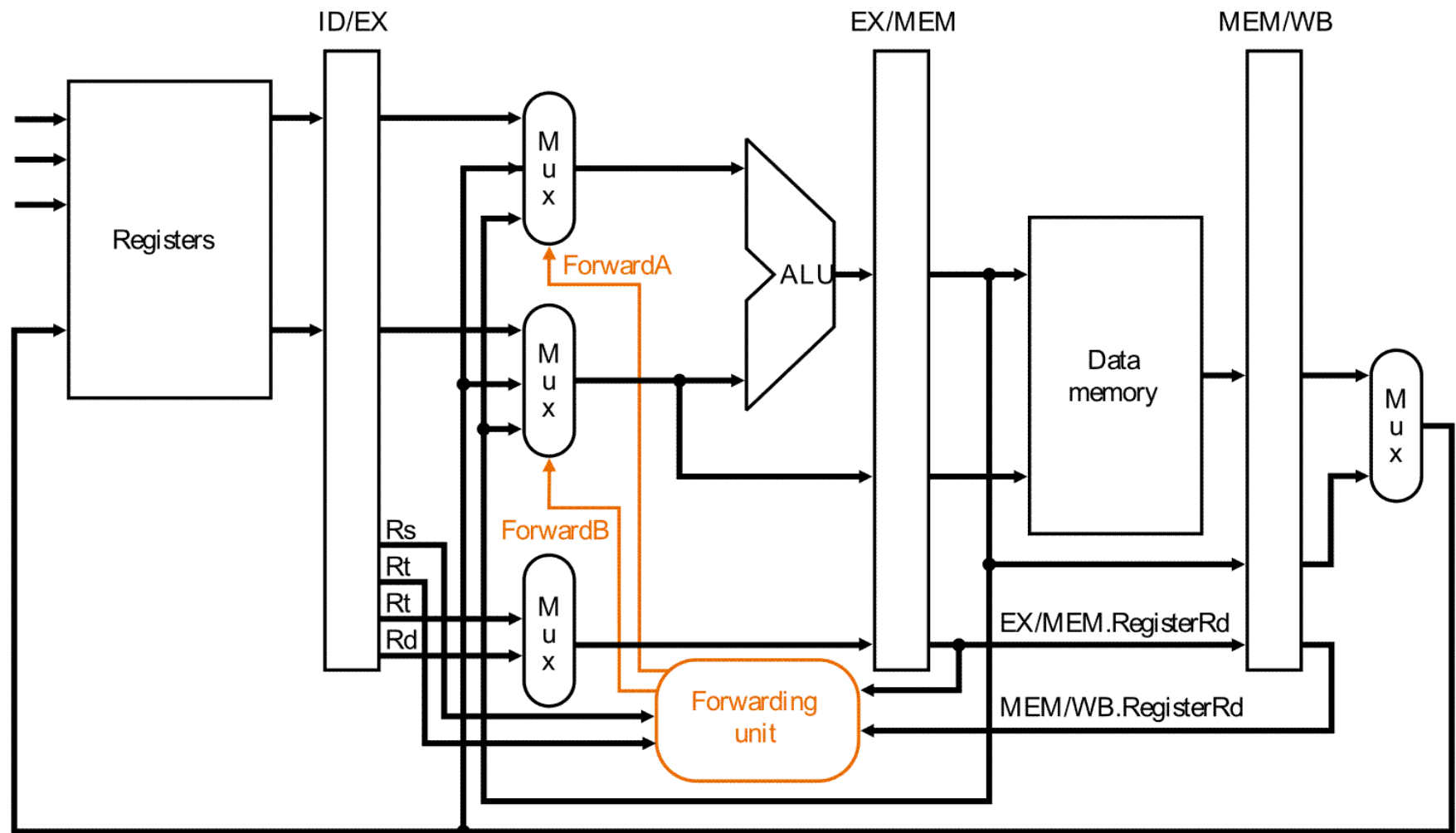
# Transfers (without Forwarding)

- Only "forward" direction and final register modification



# Transfers (with Forwarding)

- EX/MEM  $\rightarrow$  ALU
- MEM/WB  $\rightarrow$  ALU



# Forwarding

- Hardware solution to most data hazards (between EX-MEM, EX-WB stages)
- Transfer of most-up-to-date results from Ex/Mem and Mem/WB to ALU input
- Hardware: combinatorial comparators of:
  - register numbers to be modified  
(Ex/Mem.Rd lub Mem/WB.Rd)
    - with
  - register numbers of operands for ALU  
(ID/Ex.Rs lub ID/Ex.Rt)
- Destination register is always updated in program order

# Forwarding

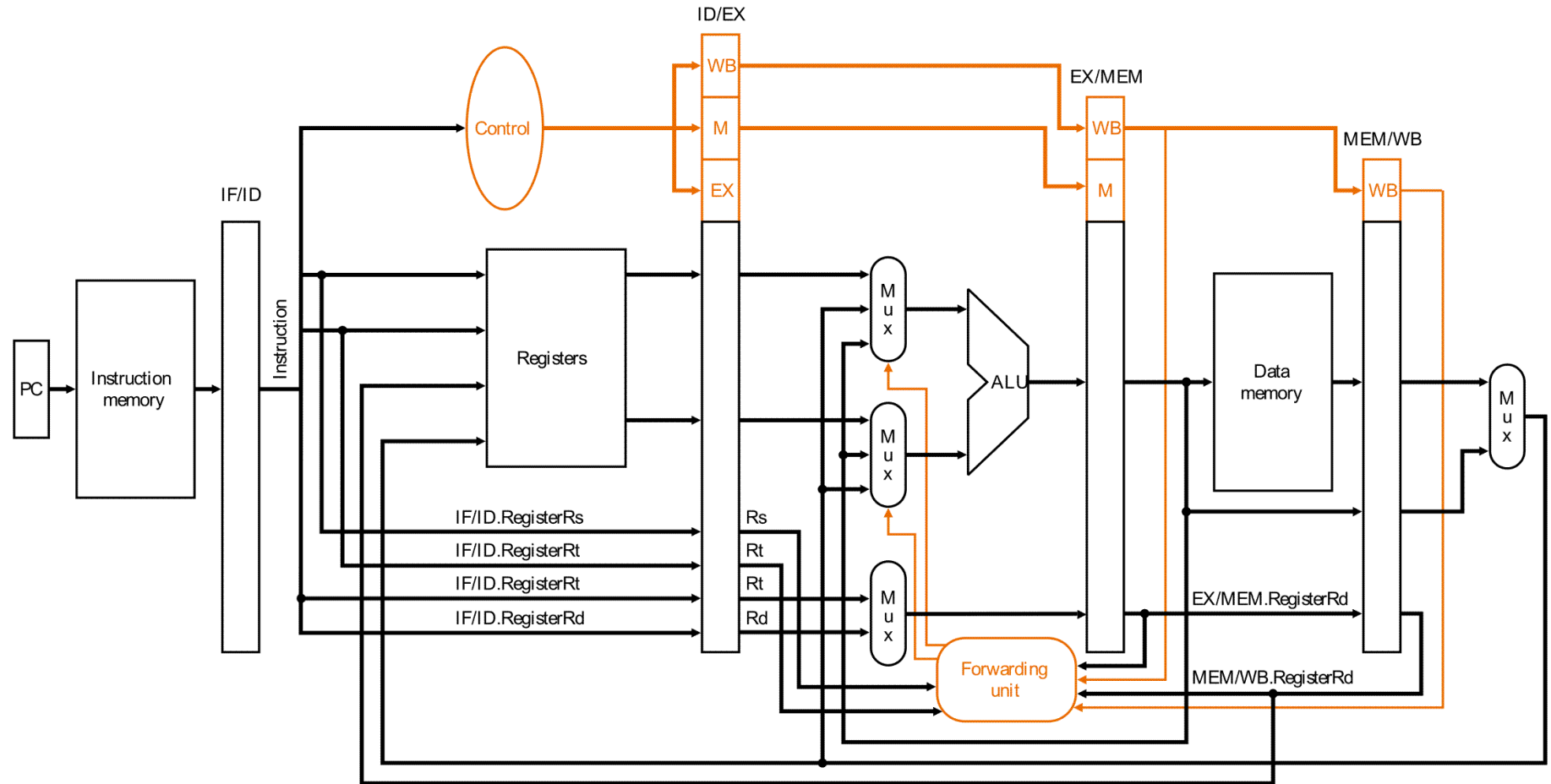
- Multiplexers at ALU input are controlled by forwarding alone (not by main control unit)

<i>Multiplexer</i>	<i>Dane</i>	<i>Opis</i>
For_A=00	ID/EX	Register File to ALU
For_A=10	EX/MEM	ALU to ALU
For_A=01	MEM/WB	MemData or ALU toALU
For_B=00	ID/EX	Register File to ALU
For_B=10	EX/MEM	ALU to ALU
For_B=01	MEM/WB	MemData or ALU toALU

- Forwarding is transparent for control unit and does not increase its complexity
- For "deep" (or parallel) pipelines, forwarding complexity grows and limits its practical application

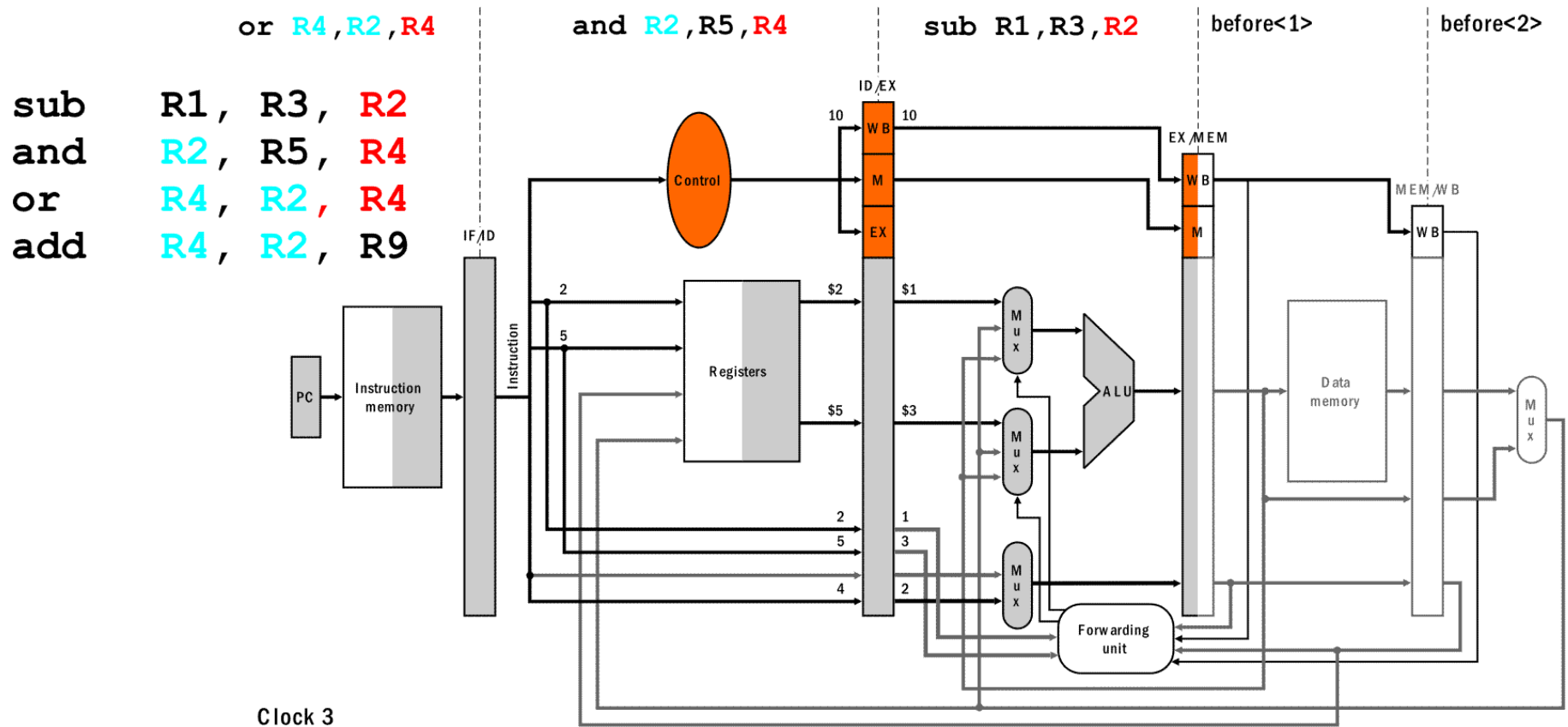


# Pipelined Architecture with Forwarding



(no jumps yet)

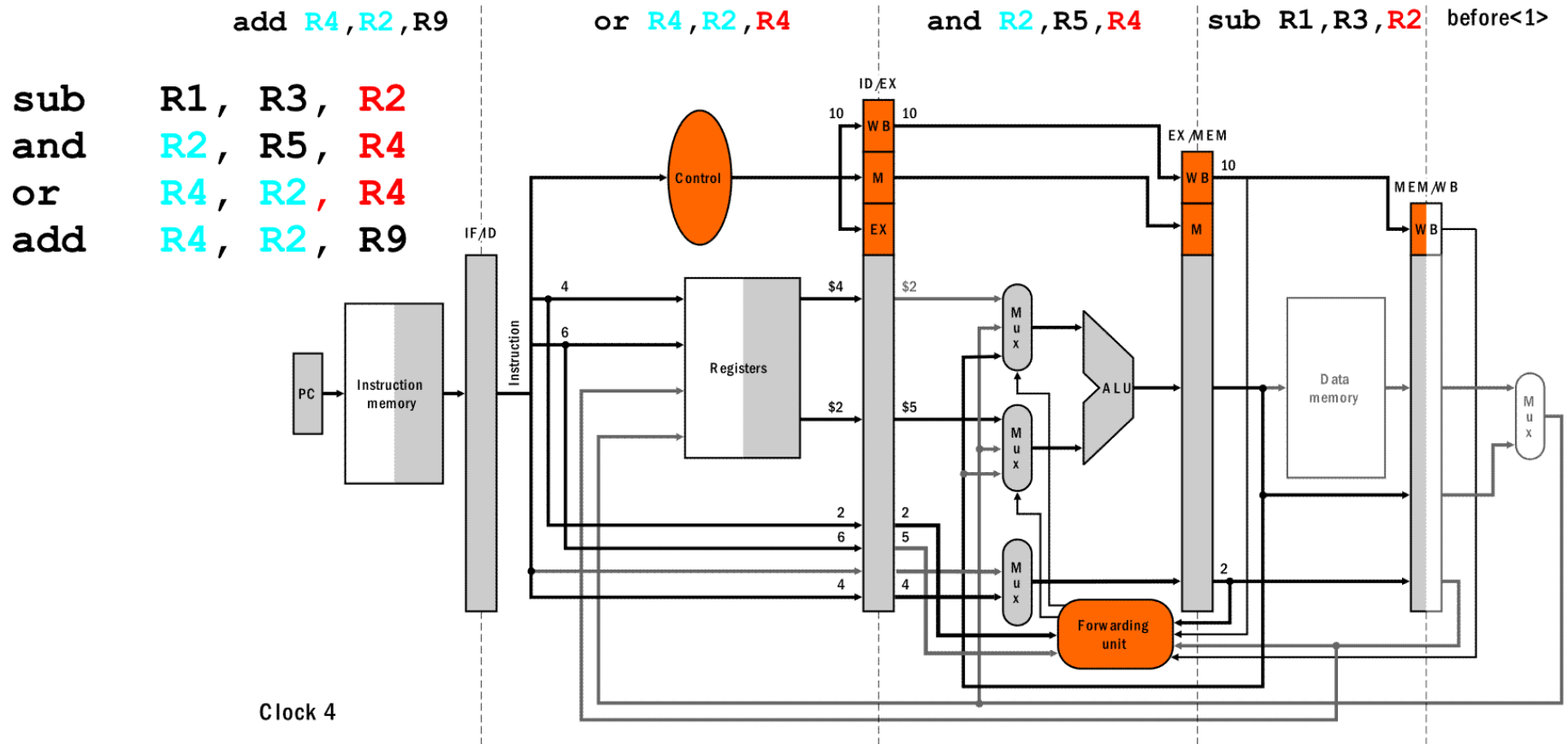
# Forwarding in action (1)







# Forwarding in action (2)



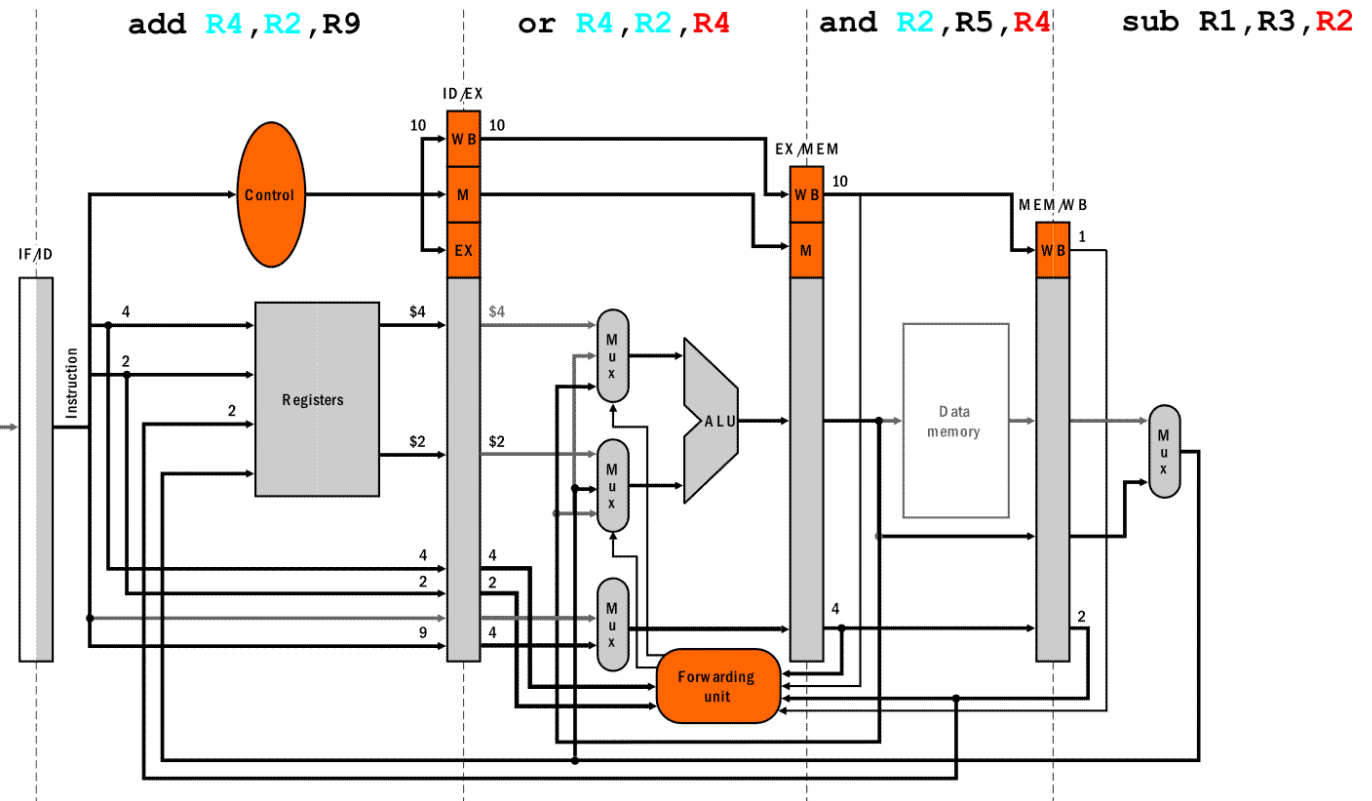


# Forwarding in action (3)

sub R1, R3, R2  
and R2, R5, R4  
or R4, R2, R4  
add R4, R2, R9

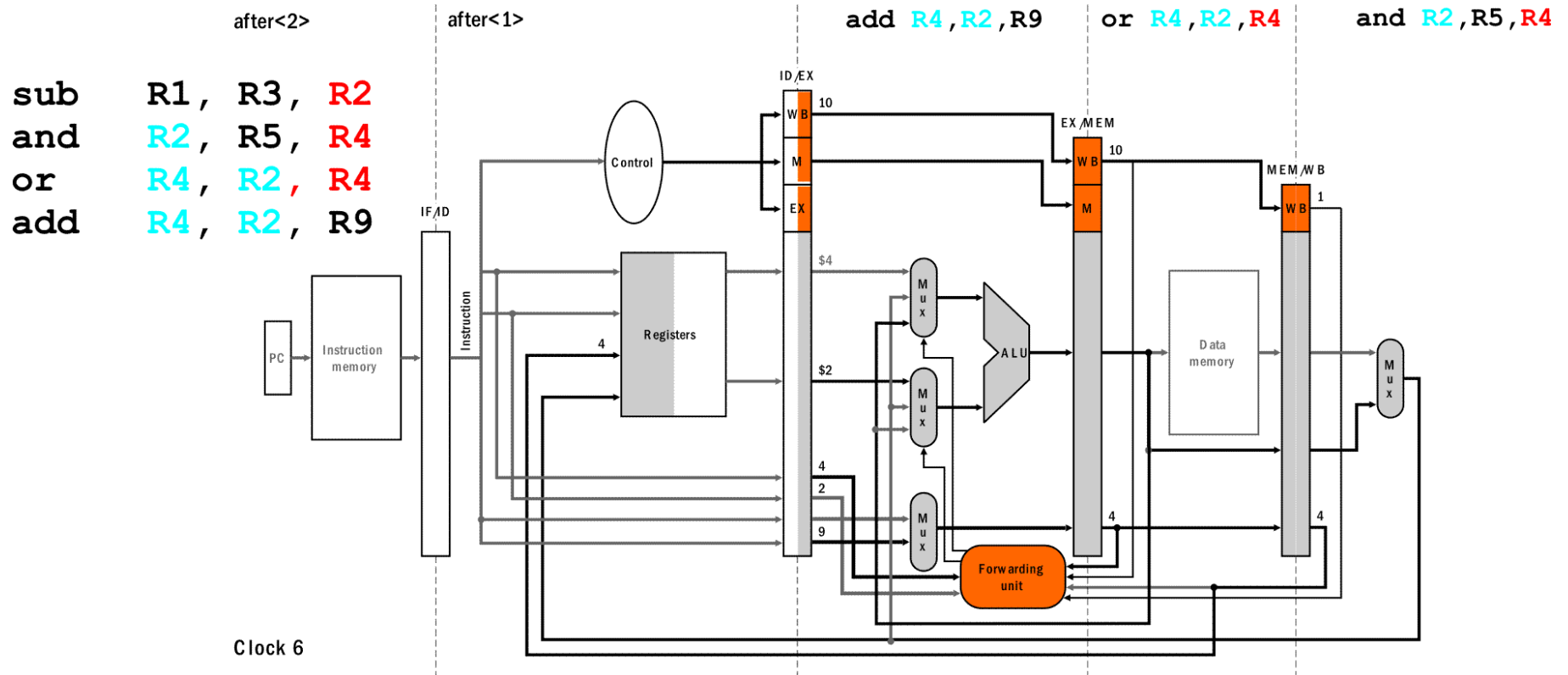
after<1>

Clock 5



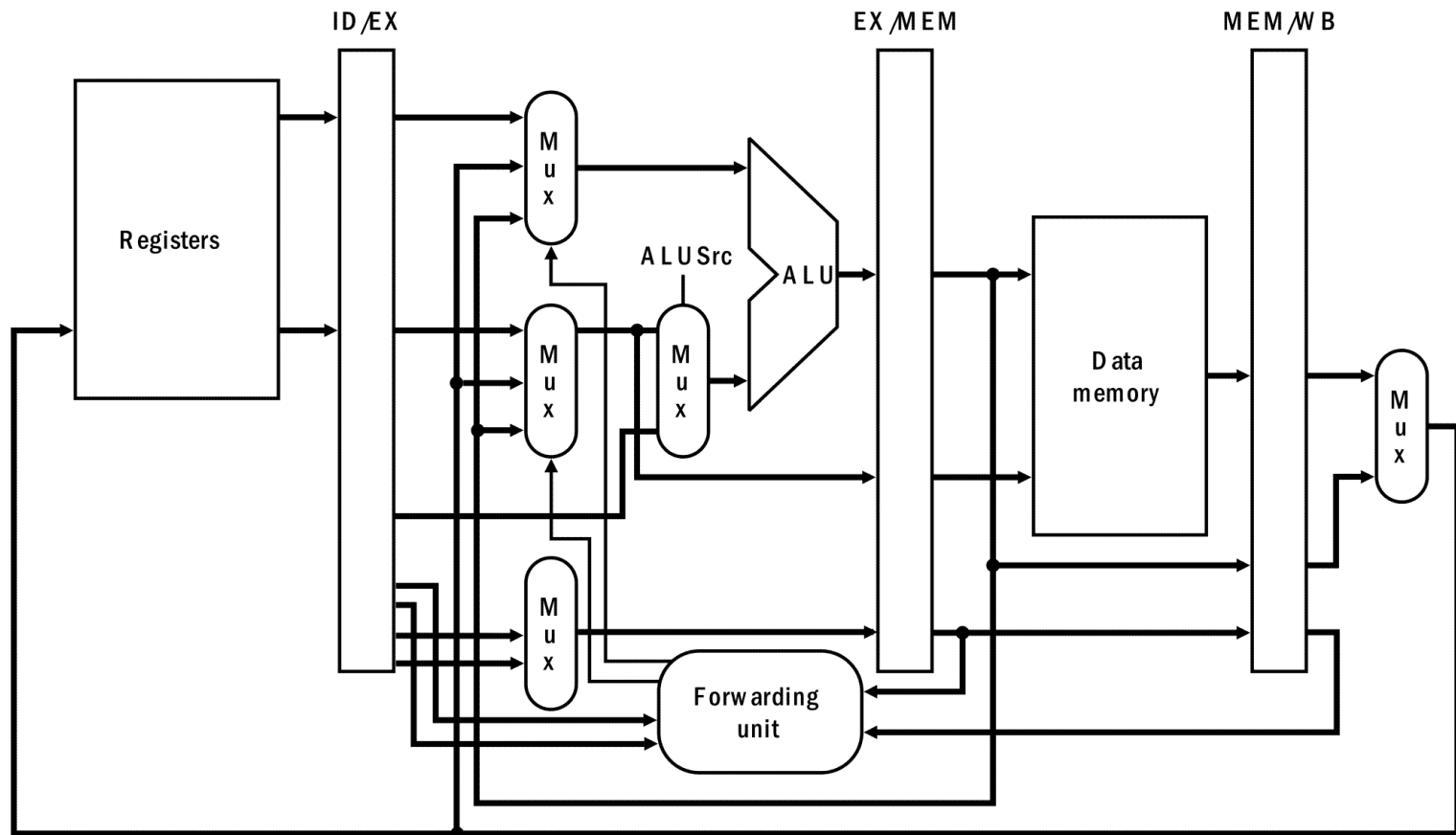


# Forwarding in action (4)



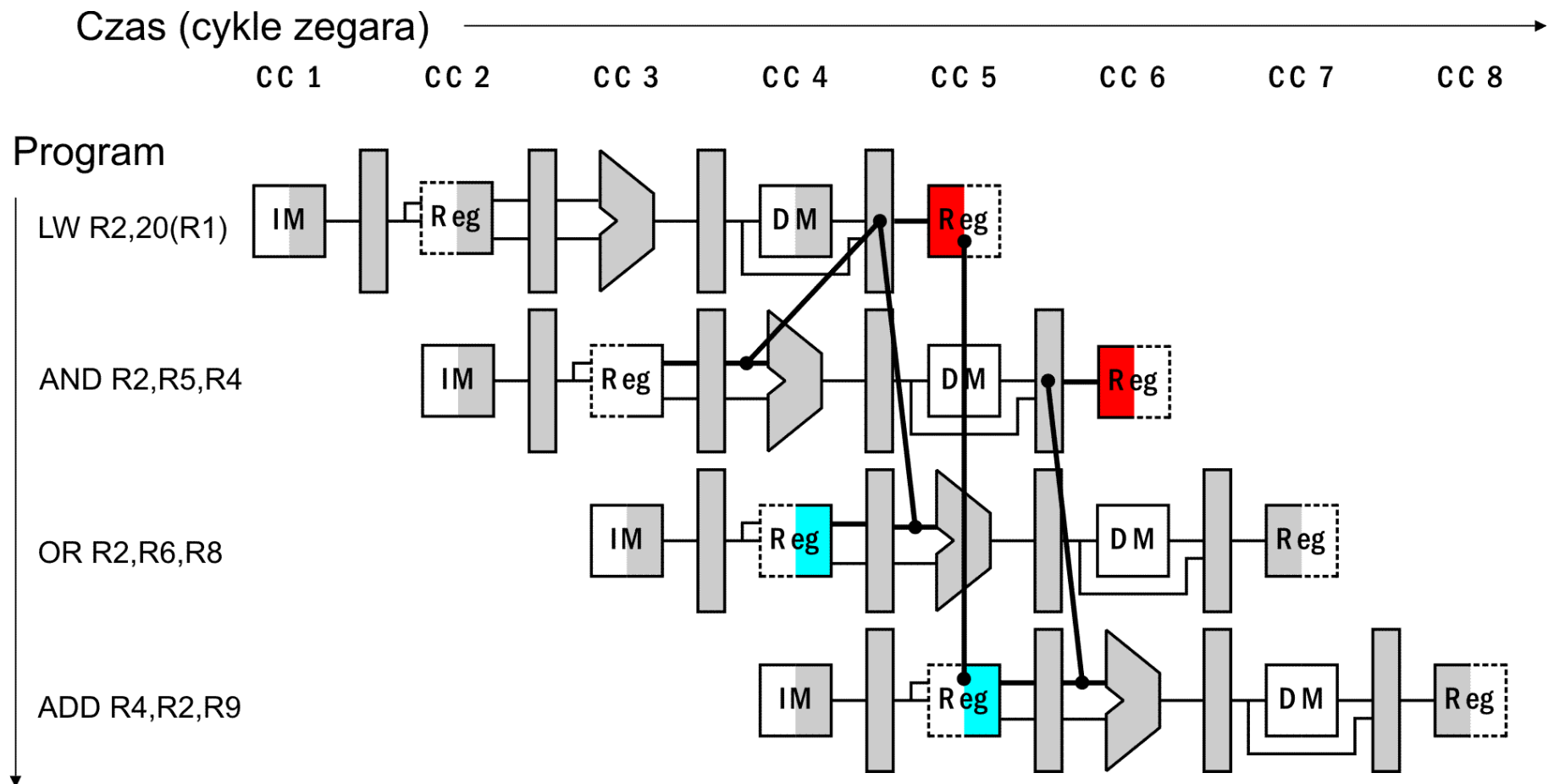
# Forwarding – ALUSrc correction

- ALUSrc is set by main control unit only
- Autonomous forwarding operation require two independent multiplexers for ALU second input



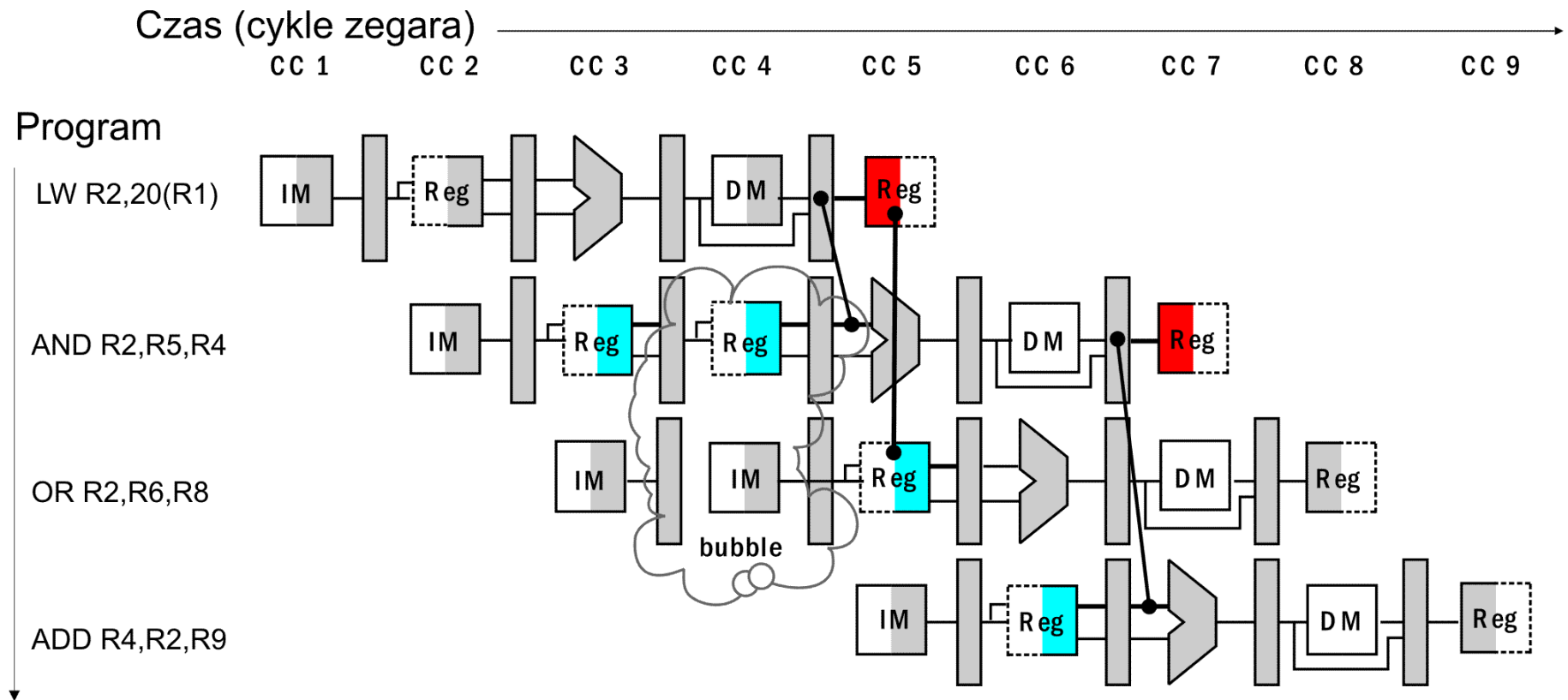
# "Hard" Data Hazards

- Forwarding cannot solve all data hazards
- e.g. *Read After Write (RAW)* – here: LW & ADD



# "Hard" Data Hazards

## Necessary pipeline stall

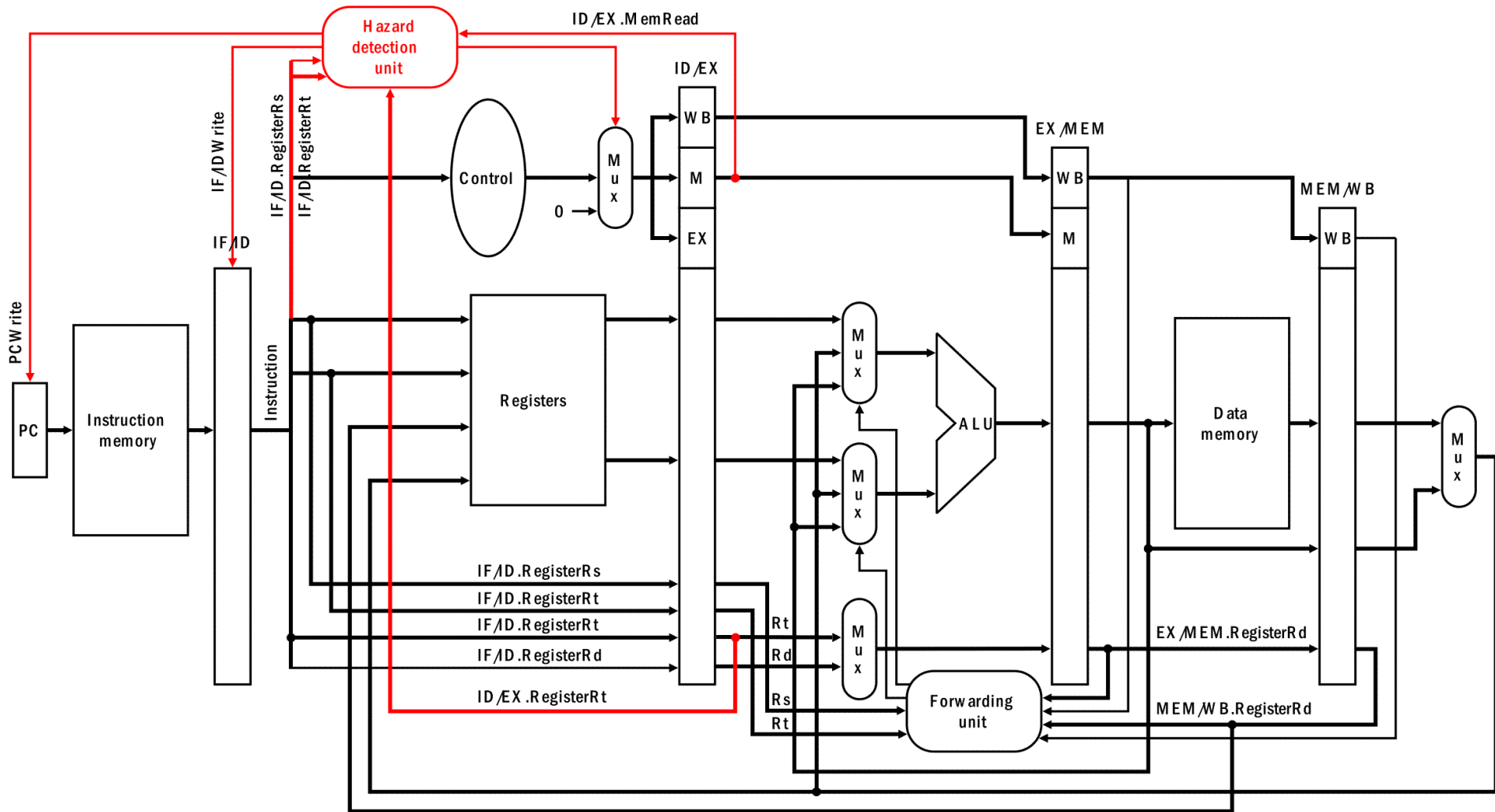




# Hardware Pipeline Stall

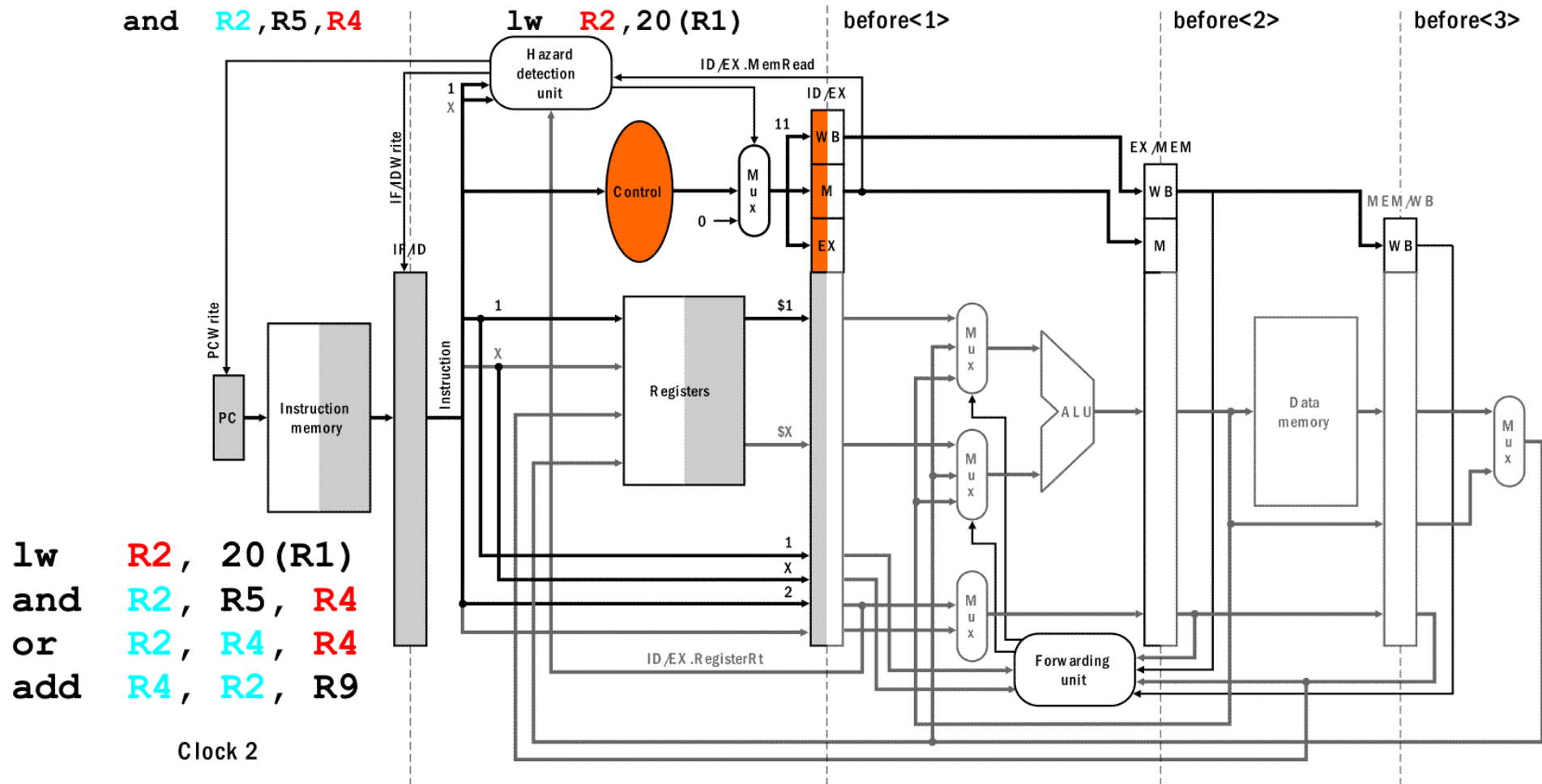
- Detection of hard data hazards must be done early (in ID)
- Additional RAW-hazard detection (combinatorial comparator) block is required in ID
- RAW-hazard detection block should be transparent for both main control and forwarding units
- RAW-hazard detects:
  - LW in stage EX (by examining ID/Ex.MemRead)
  - conflicting instruction in ID (by opcode: R-type, SW, BEQ)
  - matching numbers of registers:
    - ID/Ex.Rt (LW destination) and
    - IF/ID.Rs or IF/ID.Rt (conflicting instruction operands)

# Hardware Pipeline Stall



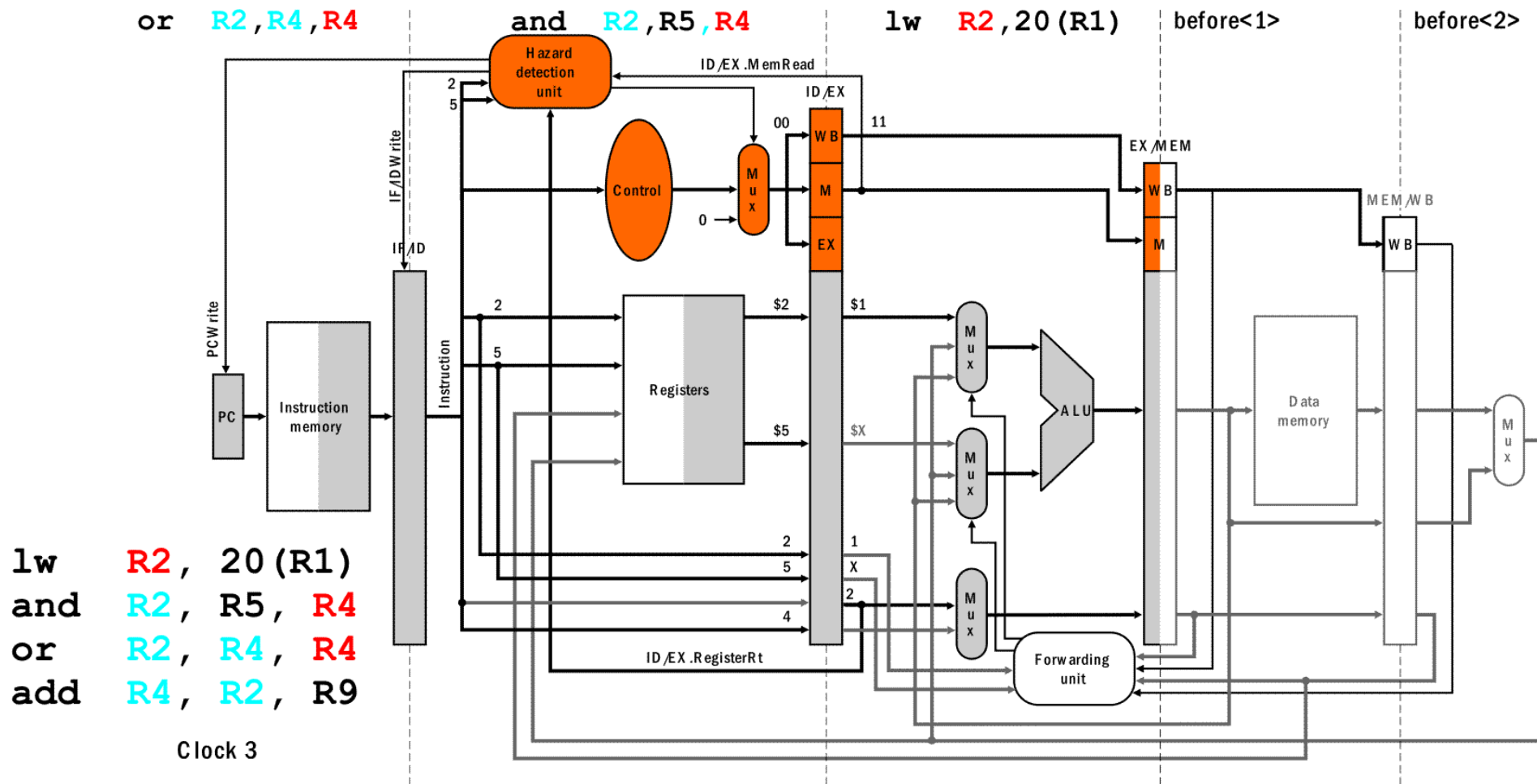


# Hardware Pipeline Stall in action (1)

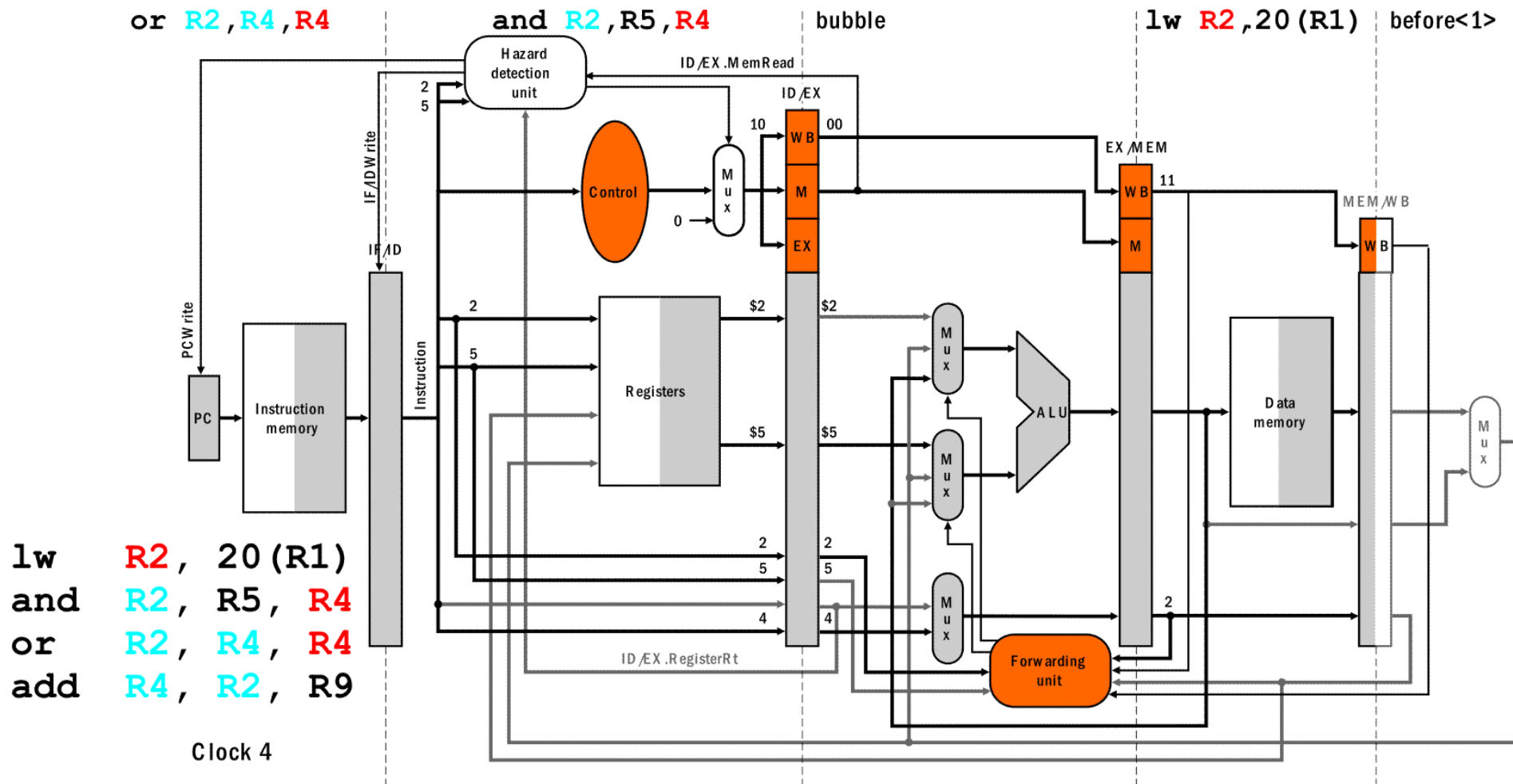


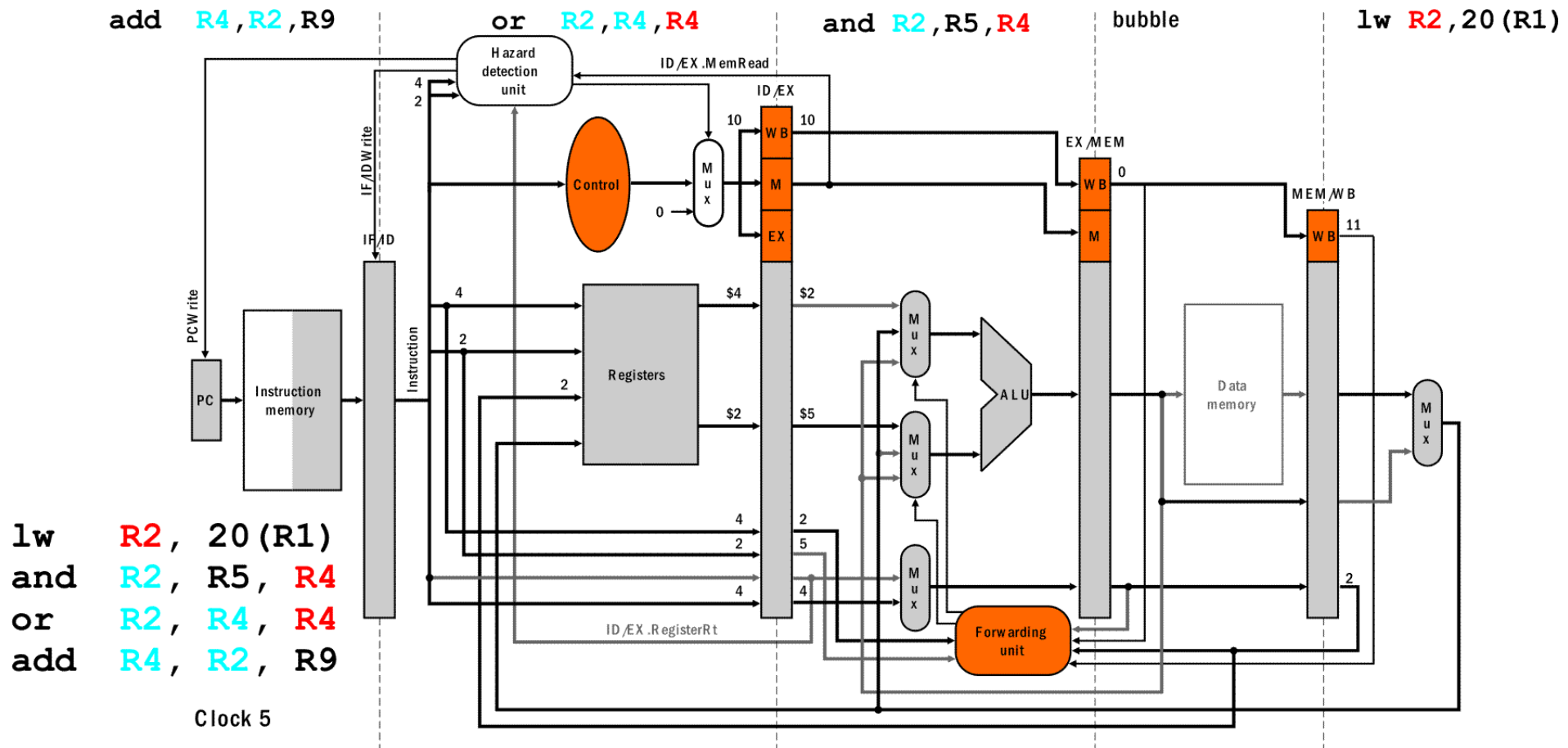


# Hardware Pipeline Stall in action (2)



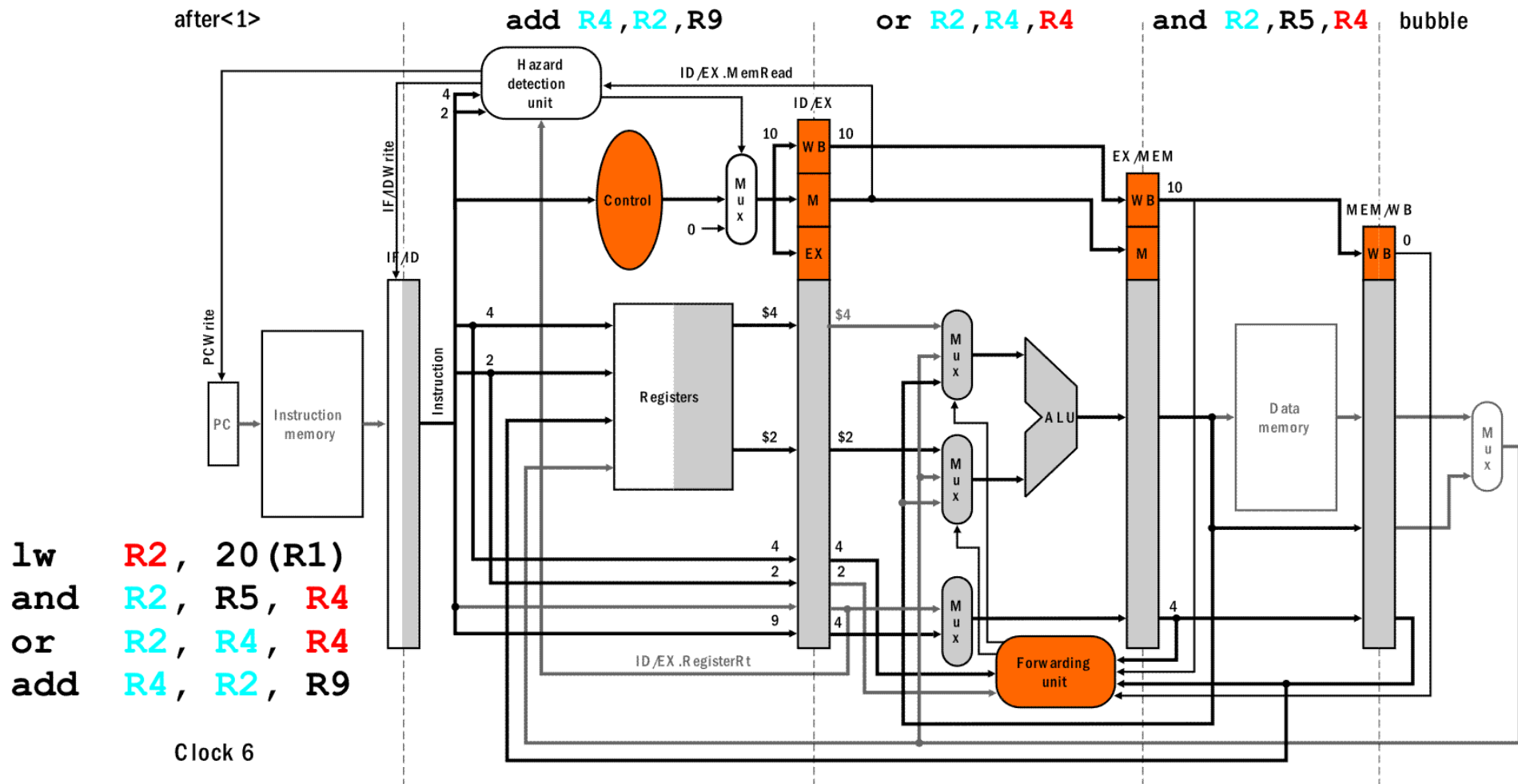
## Hardware Pipeline Stall in action (3)







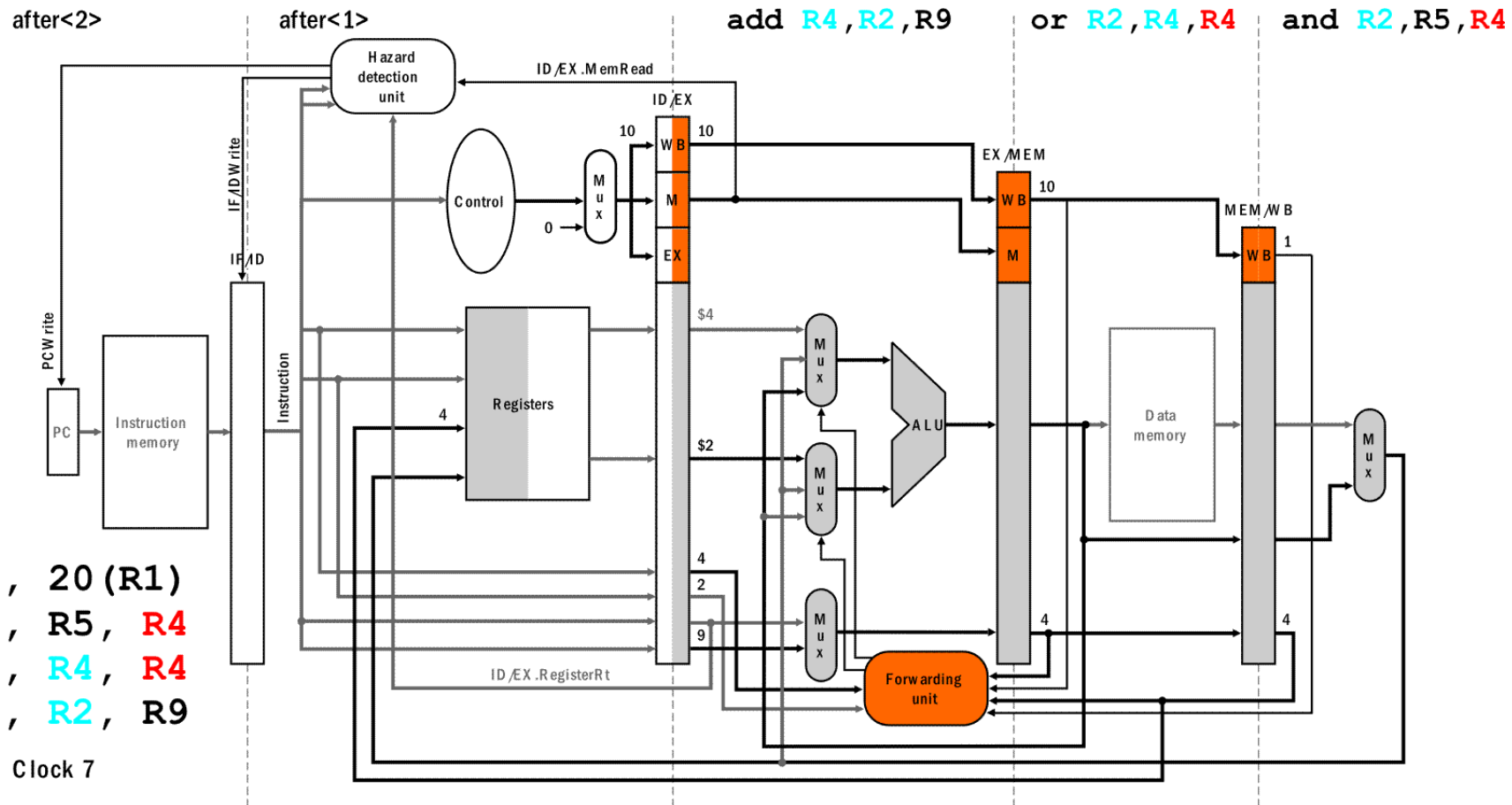
# Hardware Pipeline Stall in action (5)



# Hardware Pipeline Stall in action (6)

lw **R2**, 20(R1)  
 and **R2**, R5, **R4**  
 or **R2**, **R4**, **R4**  
 add **R4**, **R2**, R9

Clock 7

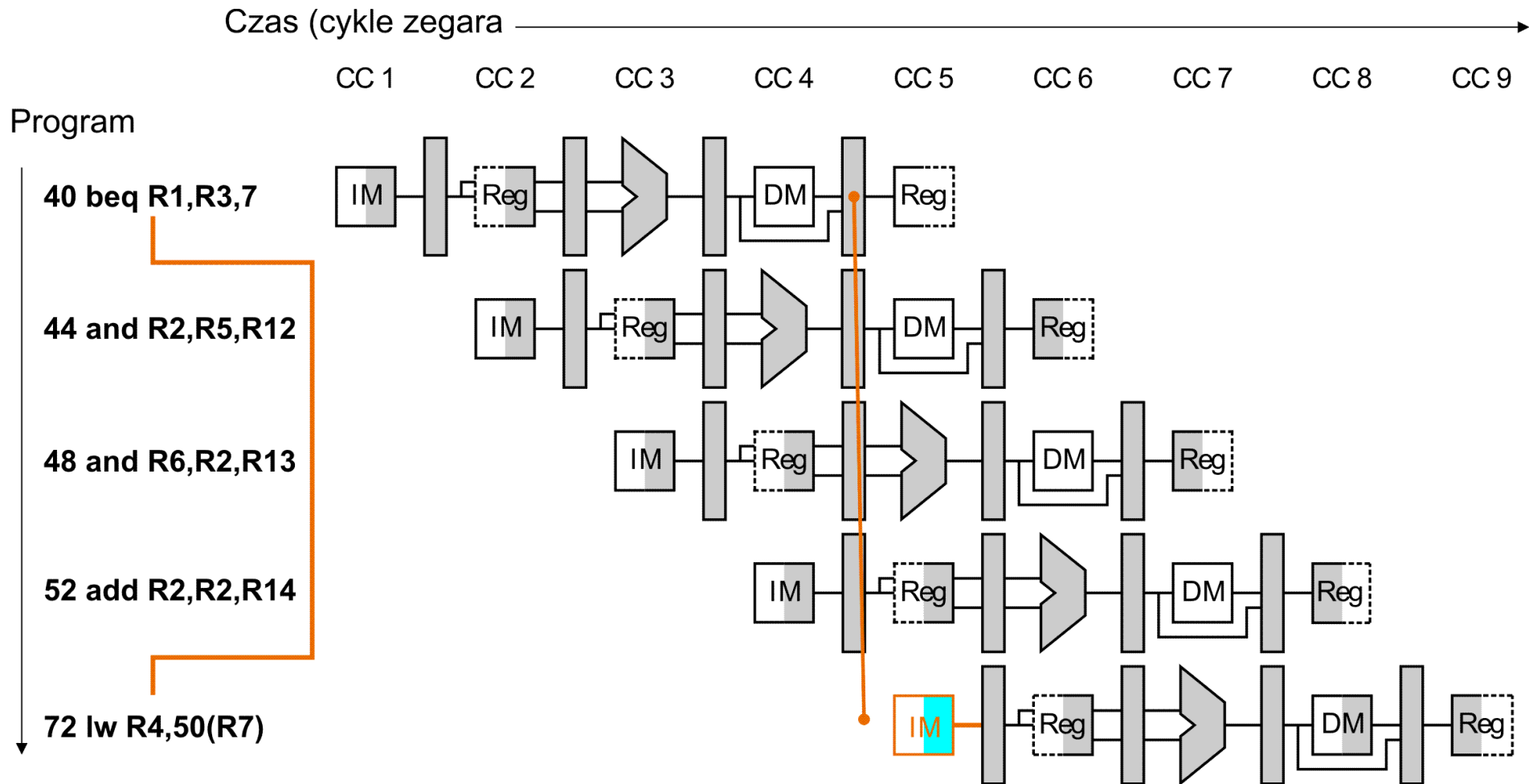


# Control Hazard

- Any jump/branch breaks the natural sequence of instructions and spoils the pipeline ( $CPI > 1$ )
- Conditional branches (apart from address calculation) must also calculate the conditions – it may take time
- Jump/Branch execution will require a few following instructions to be invalidated
- Effective solutions:
  - Early Branch Detection – requires additional hardware
  - Branch History Table – the best, but still based on guess

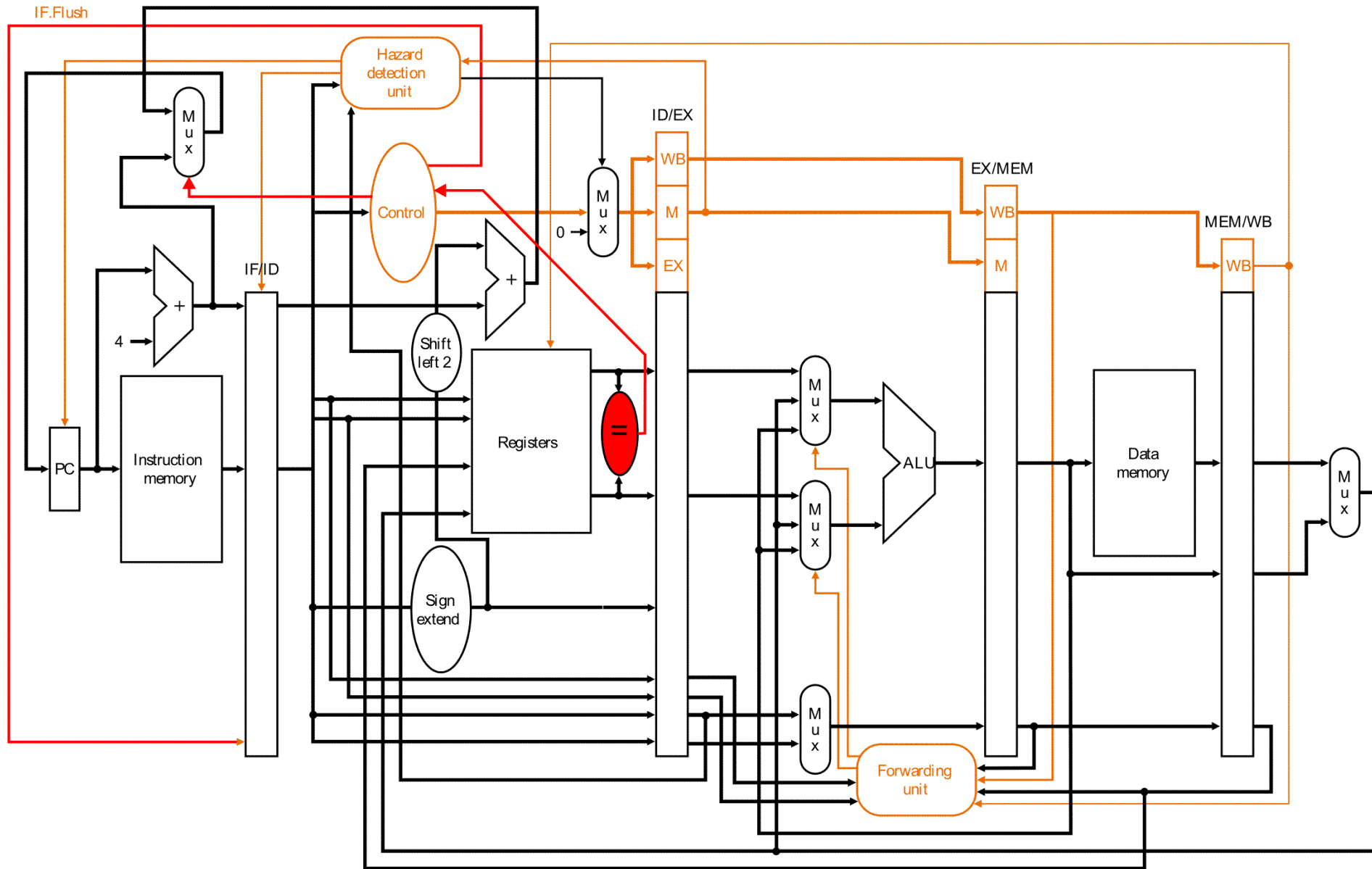
# Control Hazard

- Late branch detection (our unmodified architecture):
  - branch condition evaluated at EX, active at MEM, target instruction fetched after 3 cycles of delay





# Early Branch Detection Hardware



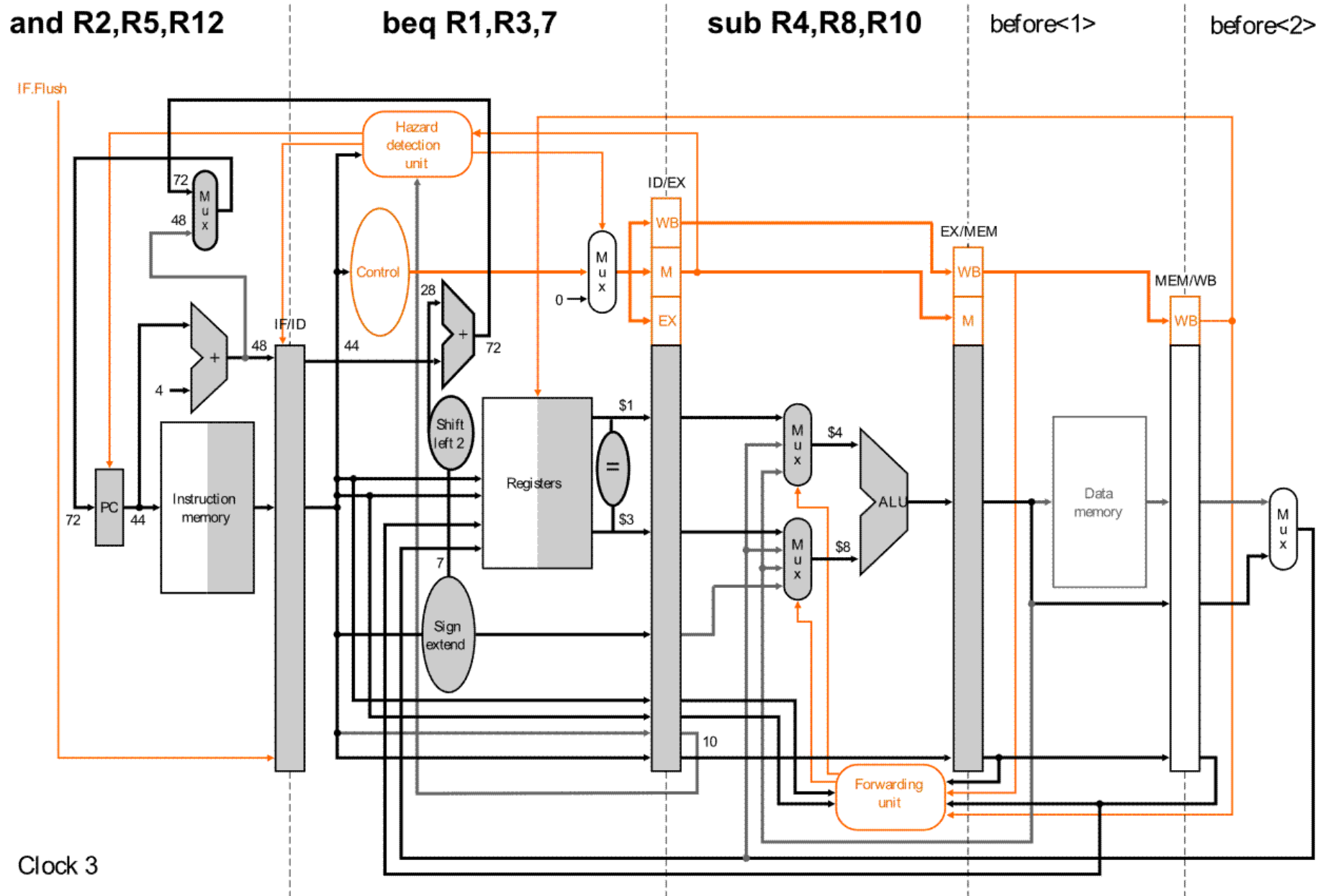
# Early Branch Detection

- Condition (simple) is calculated in ID stage – only one stage of delay will be introduced (instruction in IF)
- Only simple condition is allowed (e.g. comparison), since the registers must be read from register file
- Additional address needed in ID – dedicated for jump/branch address calculation
- Instruction in IF must be invalidated – turned into NOP (effectively the same as invalidation)
- Invalidation in IF stage ( $\rightarrow$  NOP) requires clearing the IF/ID intermediate register
  - providing, the NOP bit pattern (opcode + rest) is all 0's



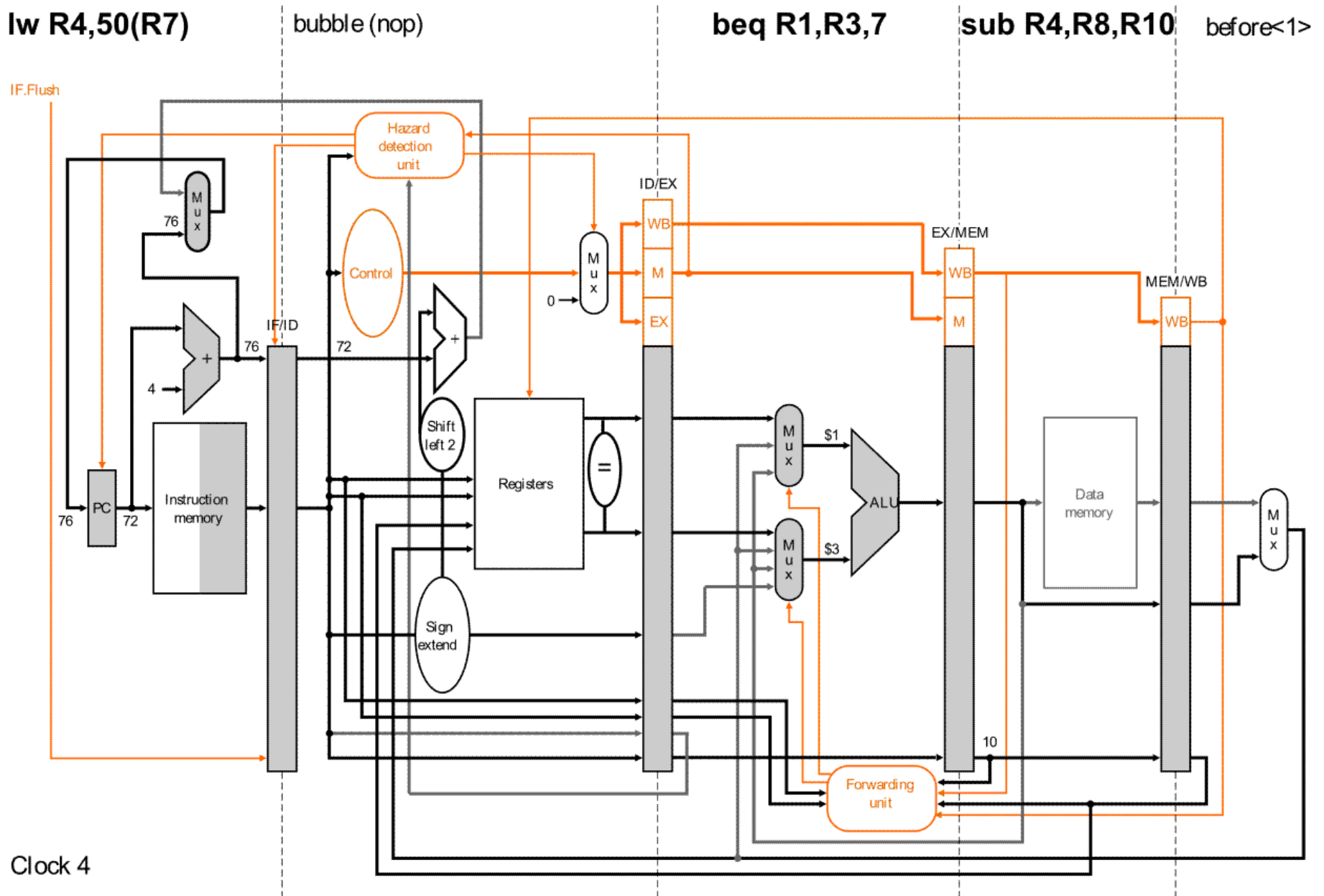


# Early Branch Detection in action (1)





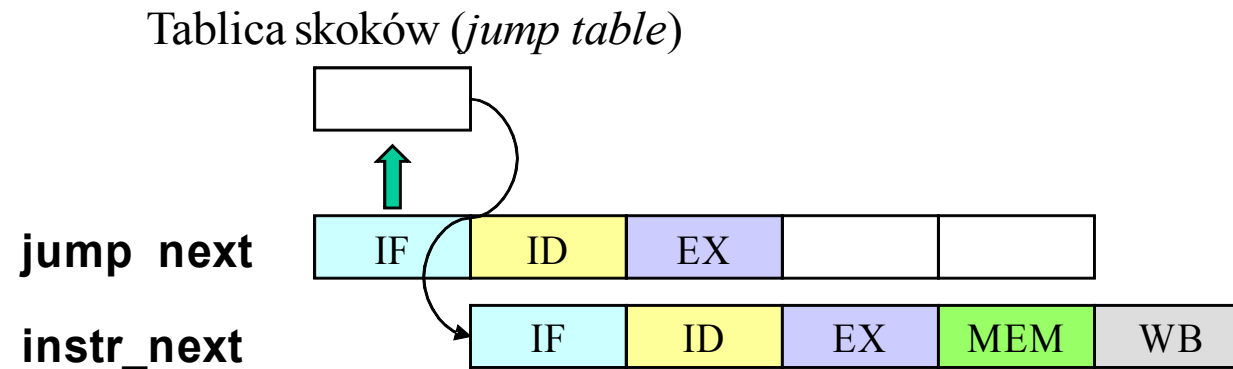
# Early Branch Detection in action (2)



Clock 4

# Branch History Table (BHT)

- BHT entry: recent branch instruction address & validated target address



- (+) No need to use early detection hardware
- (+) Complex and late condition calculation is allowed
- (+) No processing delay at all
- (-) Target is still a guess and requires validation
- (-) Misprediction causes invalidation of many instructions
- (-) Complex prediction strategies are needed (hardware)