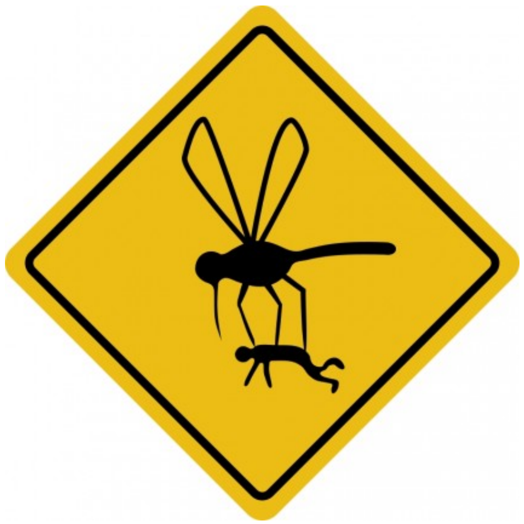
NWEN 242

Data hazards 2 - forwarding vs stalling



Agenda

- Data hazards
  - Fixing hazards with forwarding and stalling
  - Hazard detection

Forwarding

- Stalling the pipeline is bad because it reduces performance
- *Forwarding* is applying the data as soon as it is available to any units that need it. Before it is available in the register file
- Forwarding is an alternative approach to address data hazards
  - It means short-circuiting the loop from the output of the ALU to the register file,
  - Normally this loop takes in the MEM and WB stages
- We need bigger multiplexors on the ALU inputs
- We need new datapaths from MEM (and WB) directly into EX

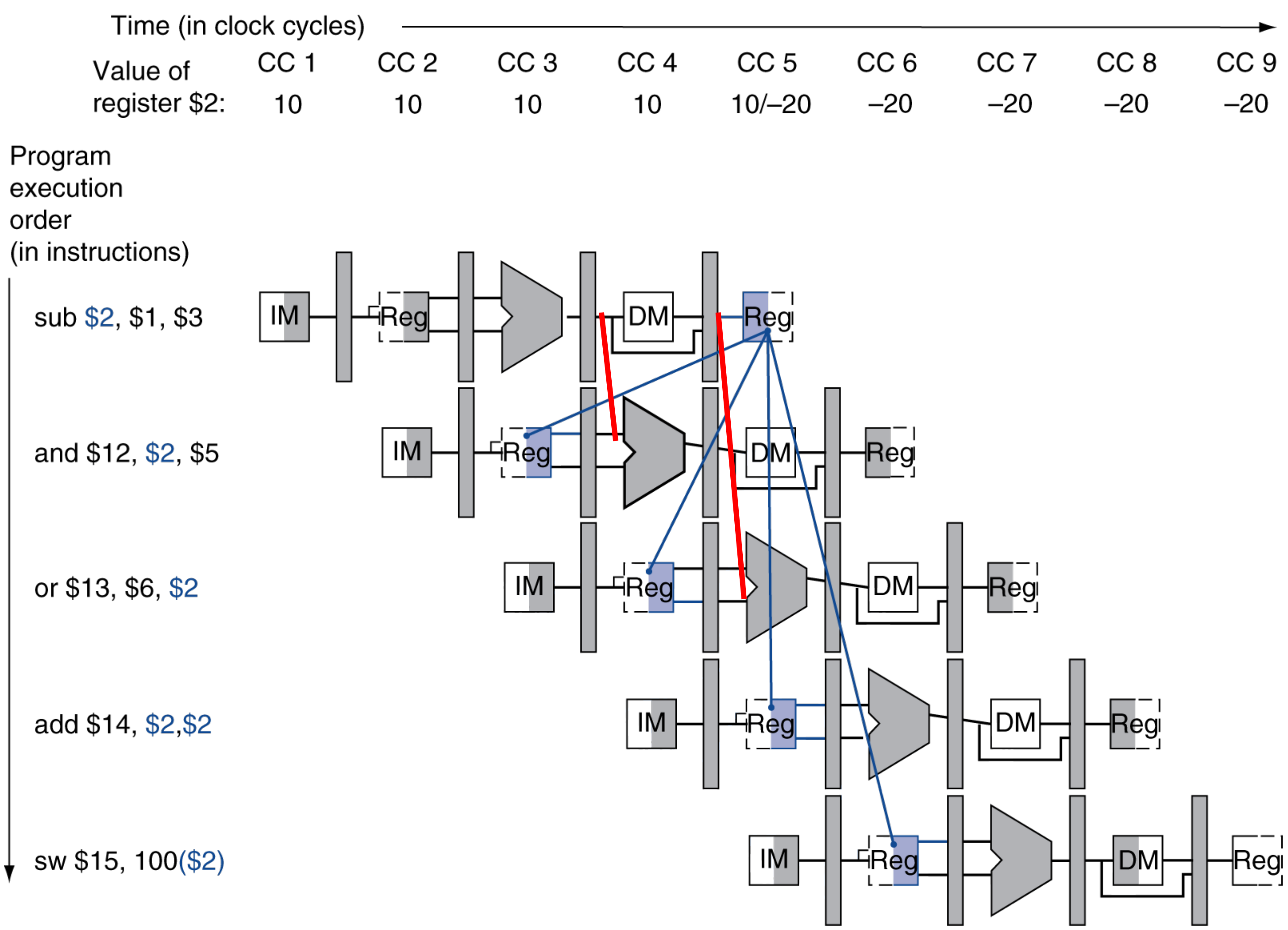
Data Hazards

Consider this sequence:

```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

- We can resolve hazards with forwarding

Dependencies & Forwarding



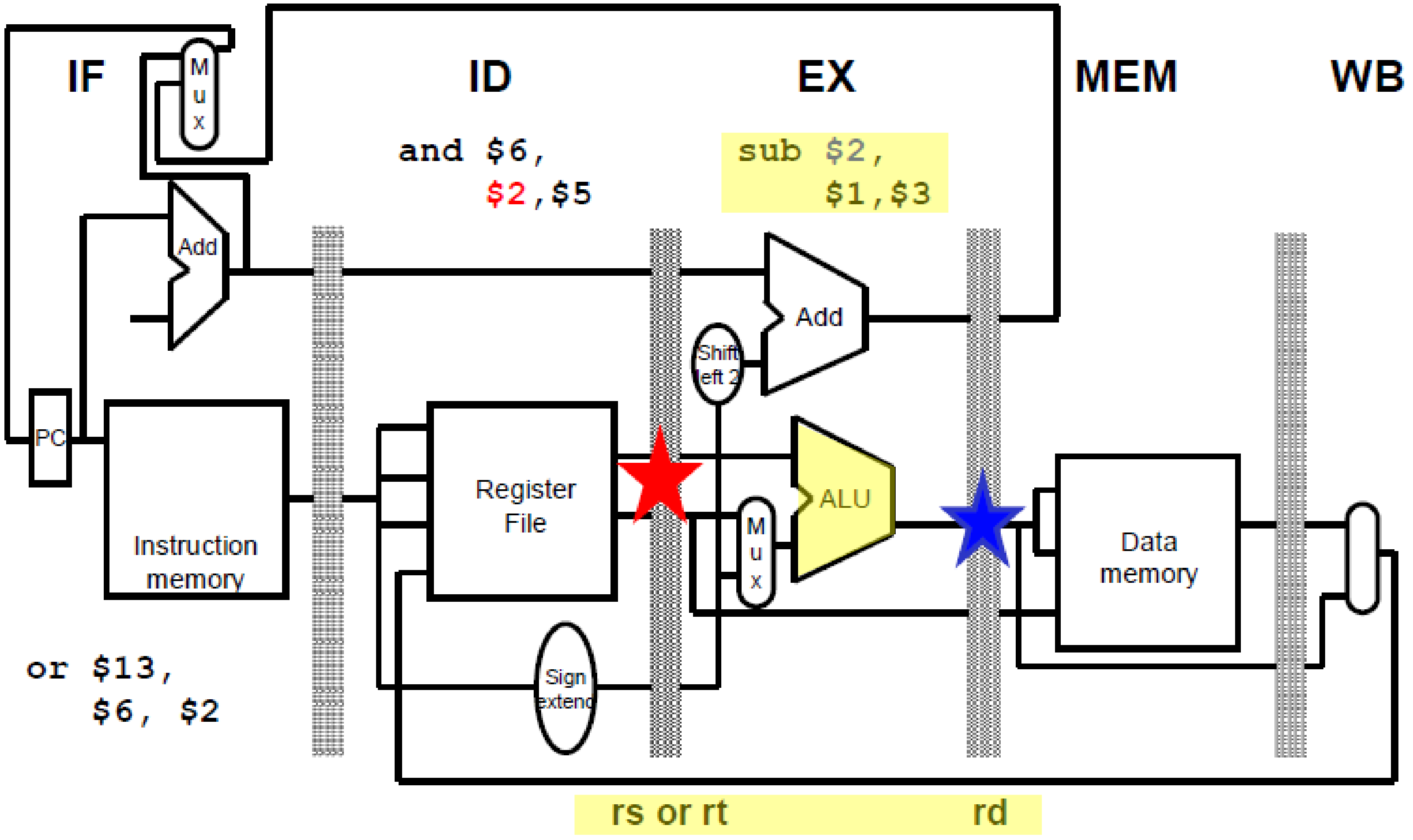
What forwarding needs?

- Forwarding needs:
  - Hazard detection, and then
  - Feeding ALU output from MEM or WB stage to ALU inputs
  - Pipeline registers contain addresses of destination registers
    - rd
- So, forwarding doesn't require any delay (nops) between two successive R-type instructions
- It passes the second instruction directly to EX stage

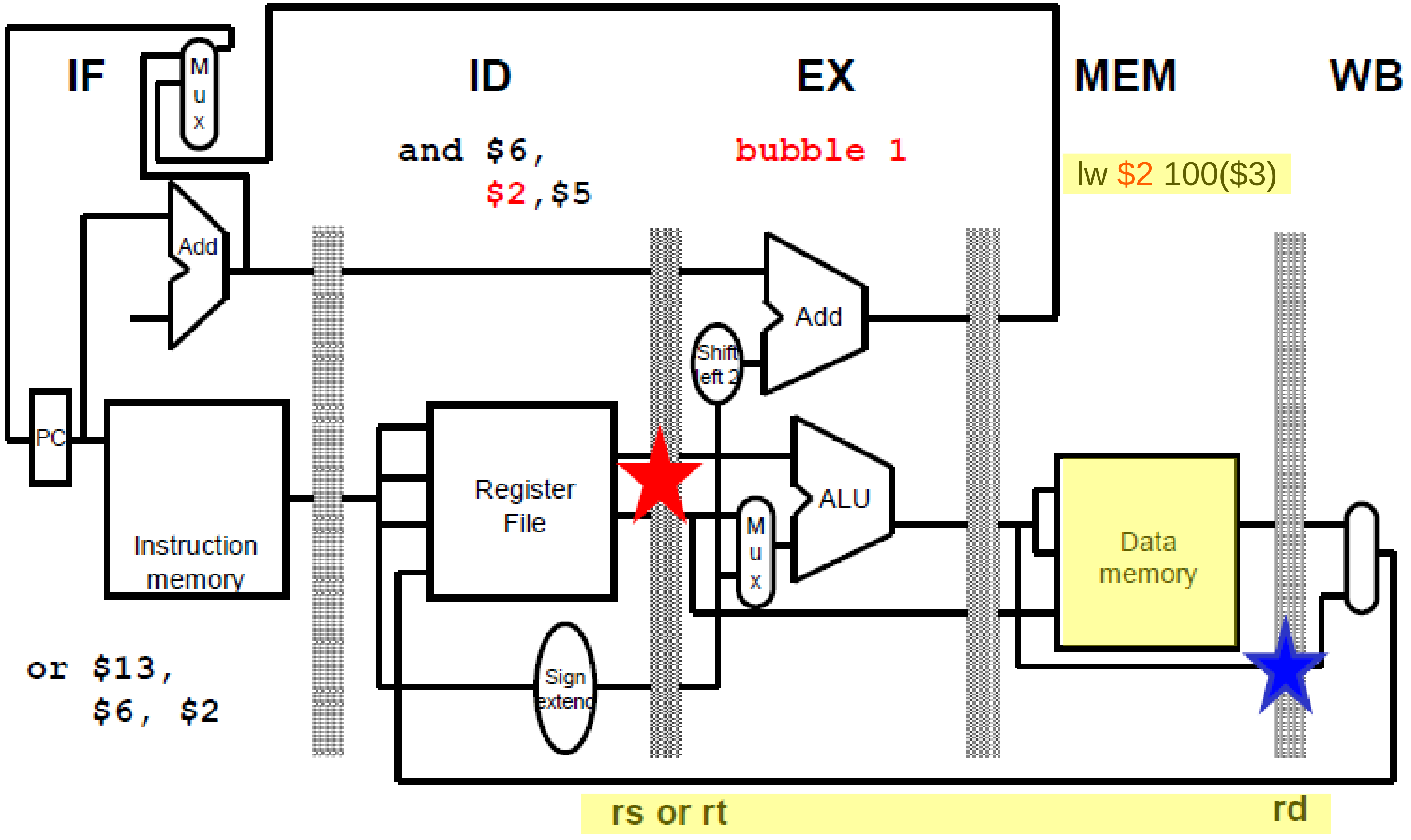
Detecting the Need to Forward

- How would you do it?

Detecting the Need to Forward



Detecting the Need to Forward



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.RegisterRs, ID/EX.RegisterRt
- Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

} Fwd from EX/MEM pipeline reg

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

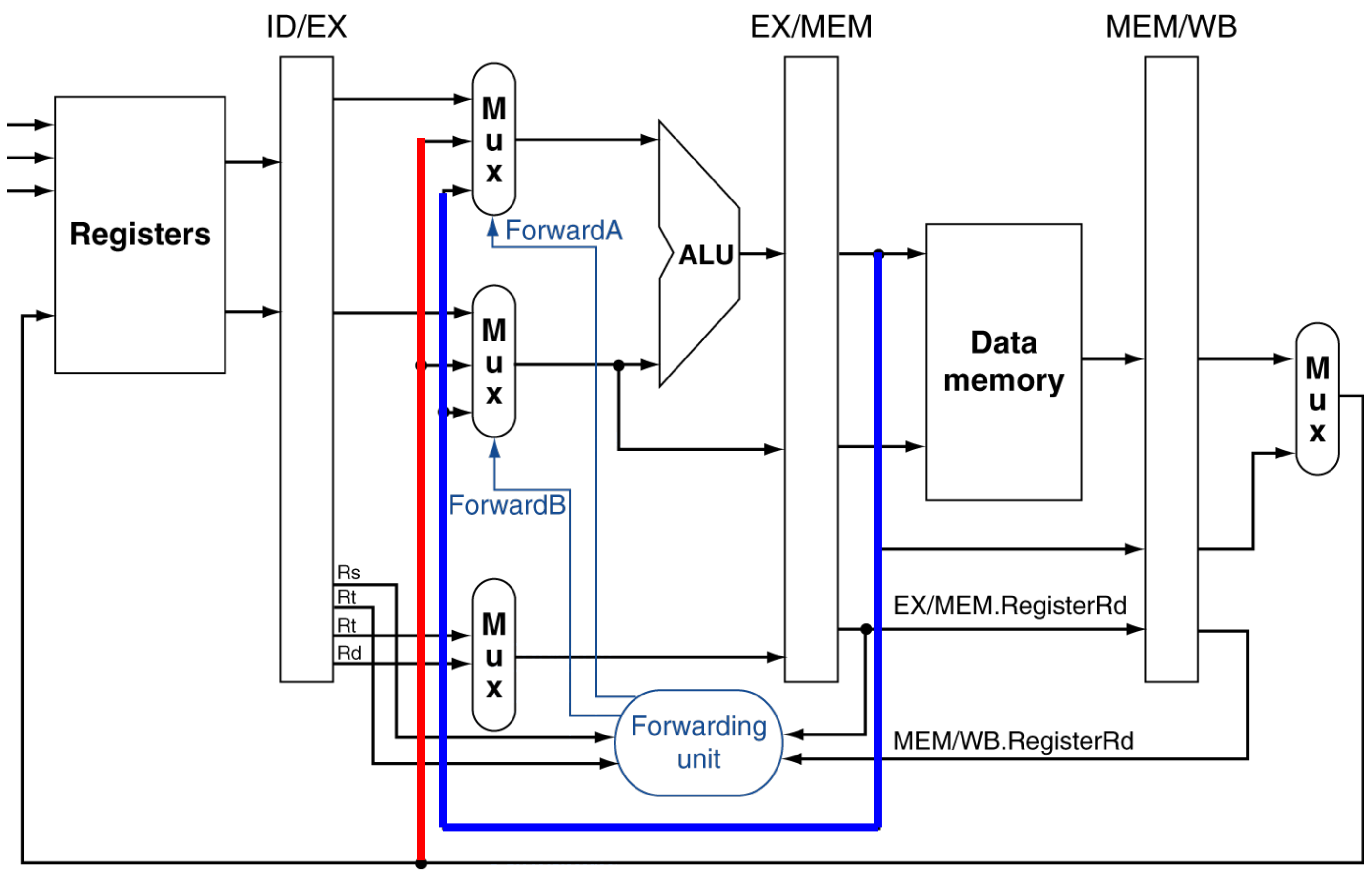
} Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite

- And only if rd for that instruction is not $zero
- EX/MEM.RegisterRd ≠ 0,
  - MEM/WB.RegisterRd ≠ 0

Forwarding Paths

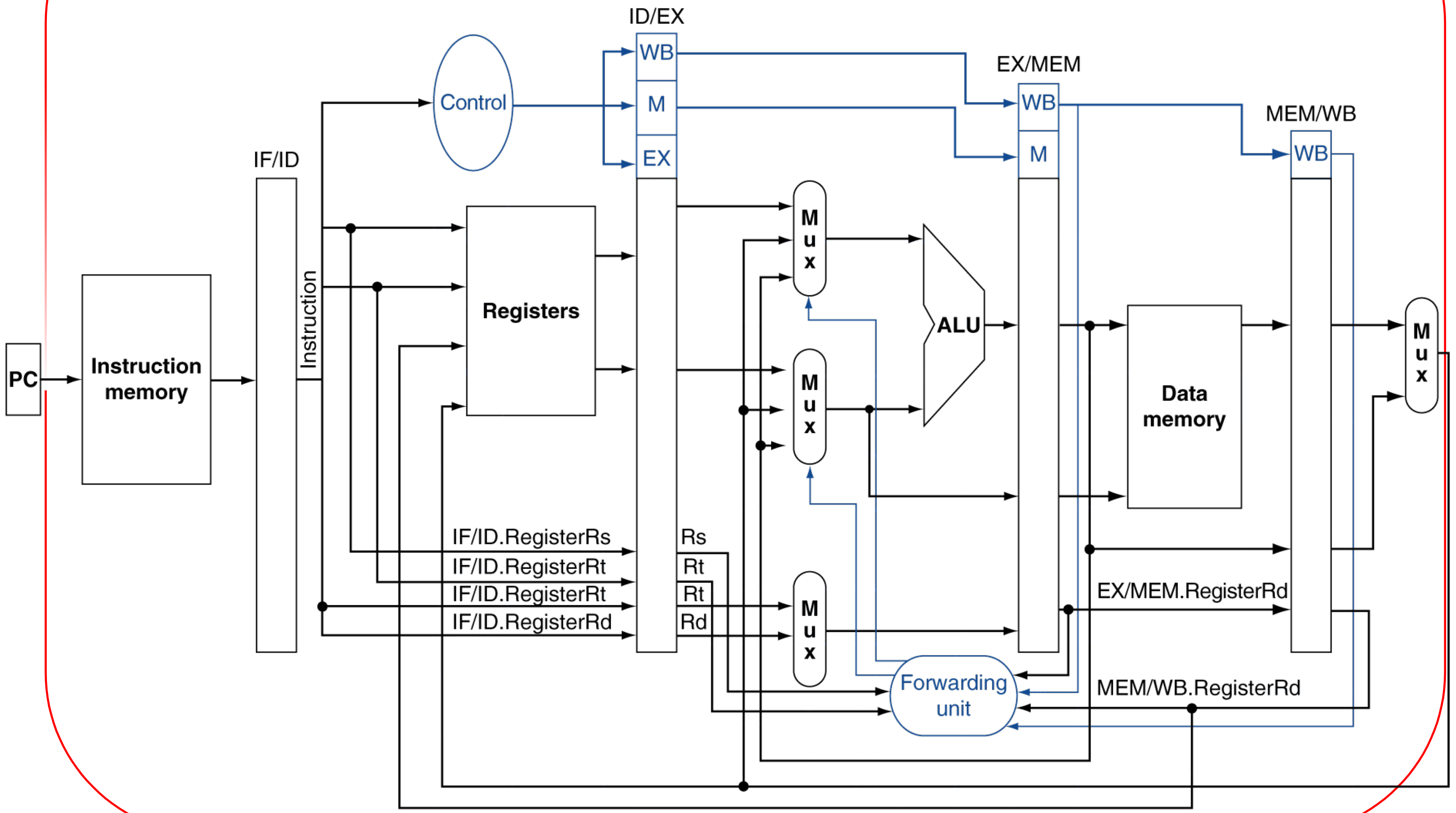


b. With forwarding

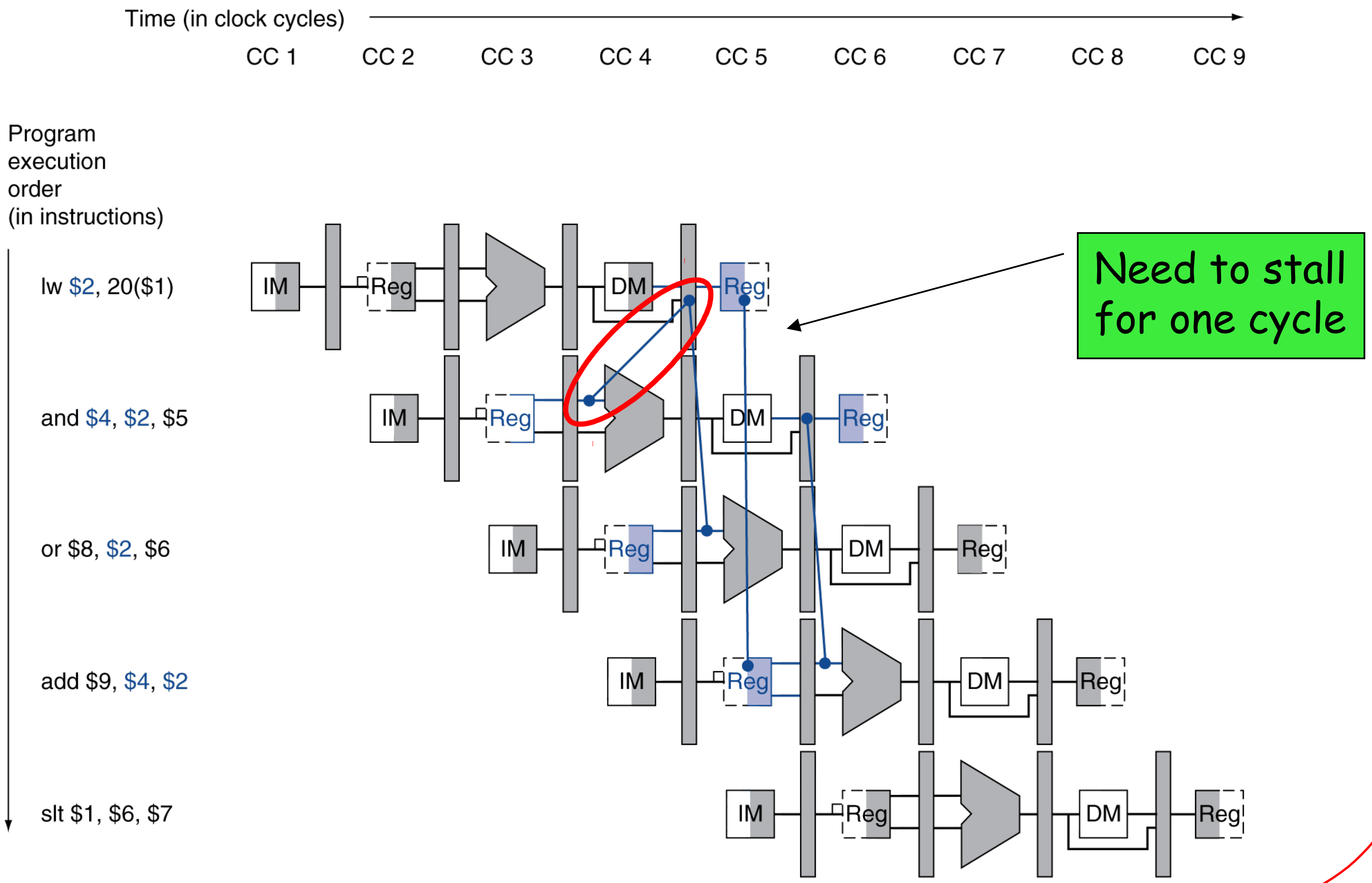
Forwarding Conditions

```
EX hazard
    if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    then    ForwardA = 10
    if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    then    ForwardB = 10

MEM hazard
    if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
        and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    then    ForwardA = 01
    if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
        and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    then    ForwardB = 01

Else    ForwardA = 00, ForwardB = 00
```

Datapath with Forwarding



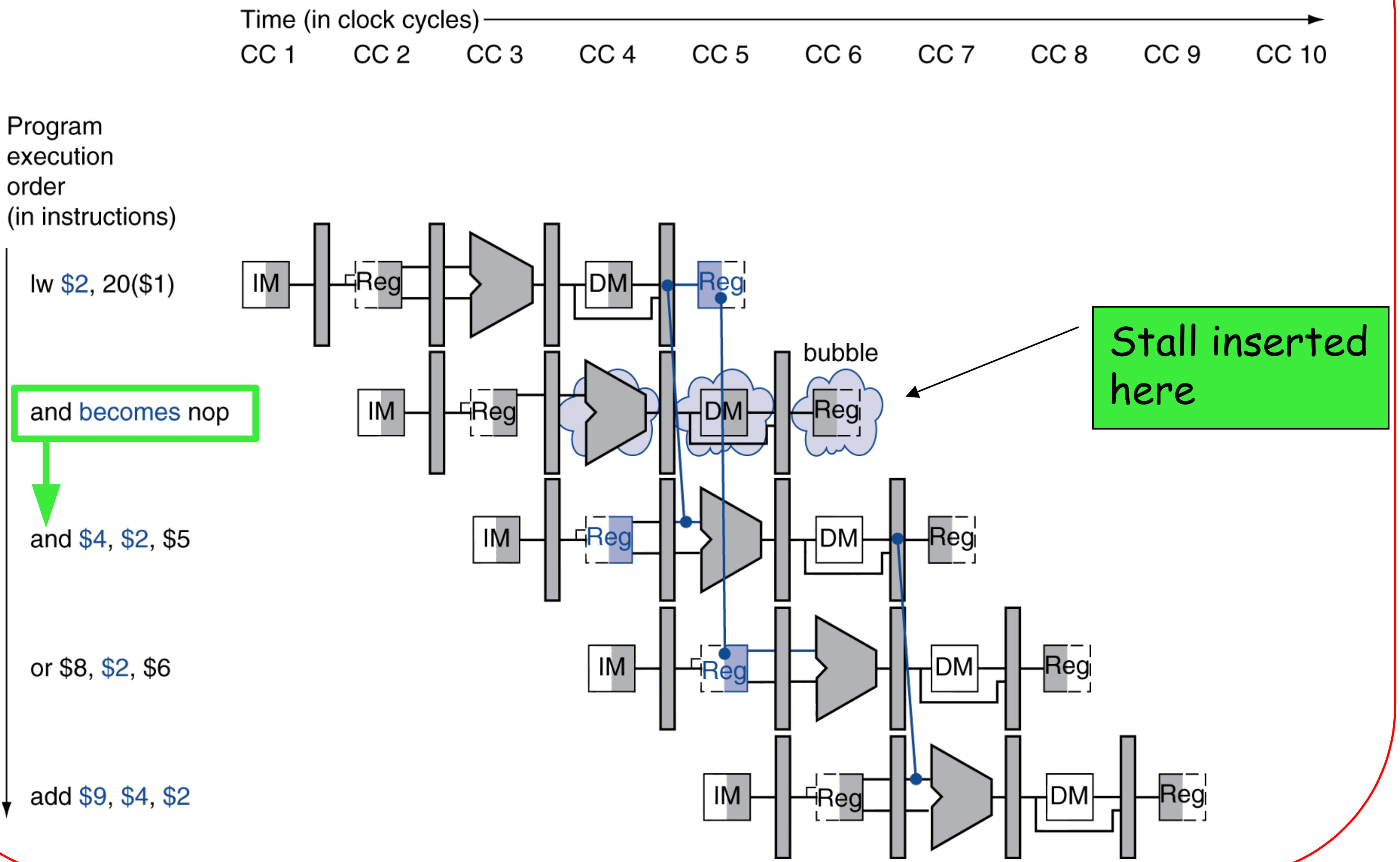
Load-word Data Hazard



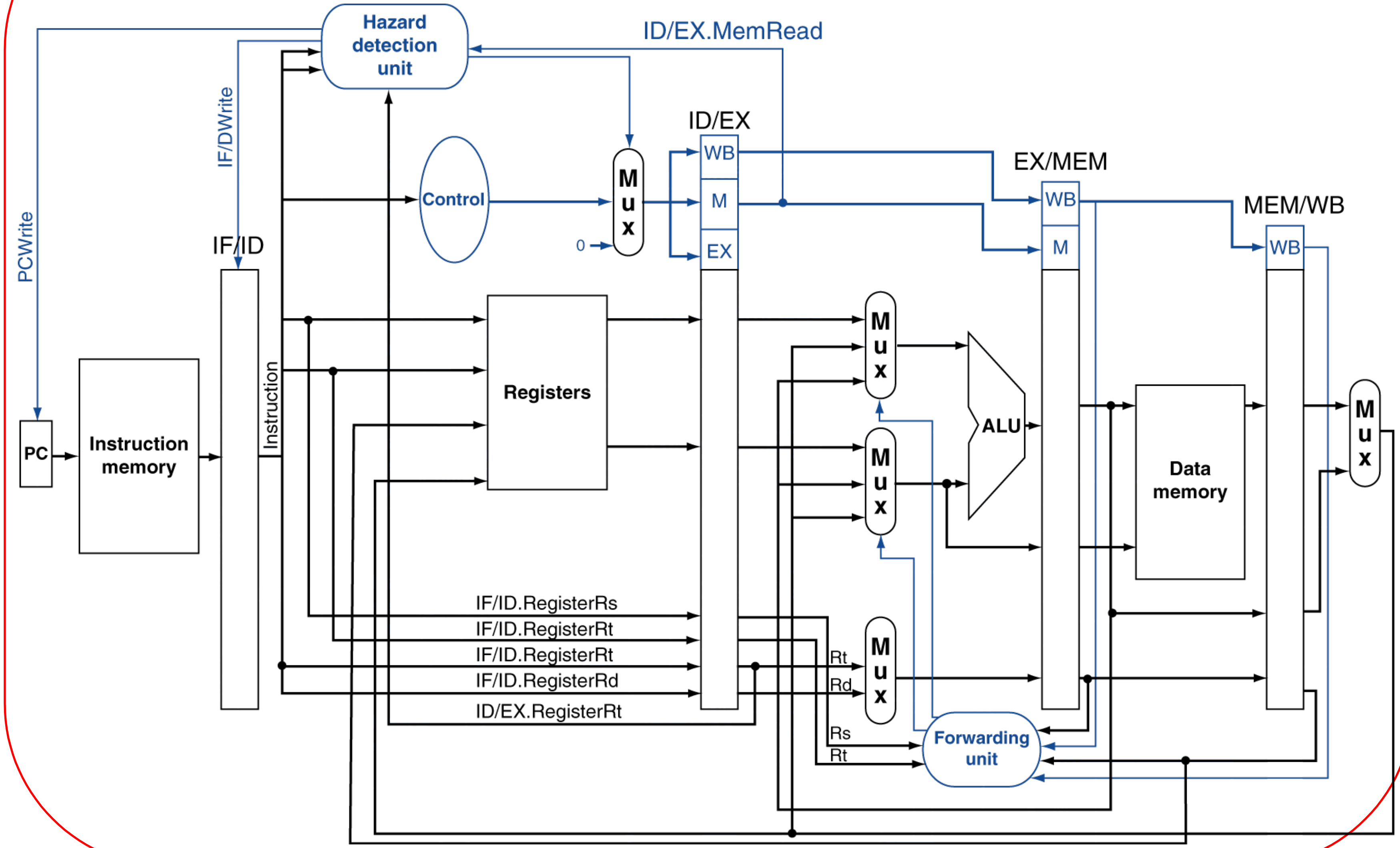
How to Stall the Pipeline (recap)

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction (ID) is decoded again
  - Following instruction (IF) is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline



Datapath with Hazard Detection



Stalls and Performance

Stalls by inserting nop bubbles into the pipeline reduces performance
- But are required to get correct results

However,
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F;

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add      $t3, $t1, $t2
sw  $t3, 12($t0)
lw   $t4, 8($t0)
add      $t5, $t1, $t4
sw  $t5, 16($t0)
```

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F;

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
                    ← stall
add     $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
                    ← stall
add     $t5, $t1, $t4
sw  $t5, 16($t0)
```

13 cycles total

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F;

```
        lw   $t1, 0($t0)
        lw   $t2, 4($t0)
stall →
        add     $t3, $t1, $t2
        sw   $t3, 12($t0)
        lw   $t4, 8($t0)
stall →
        add     $t5, $t1, $t4
        sw   $t5, 16($t0)
```

13 cycles total

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add     $t3, $t1, $t2
sw   $t3, 12($t0)
add     $t5, $t1, $t4
sw   $t5, 16($t0)
```

11 cycles total

Summary

- Forwarding unit
- Hazard detection unit
- Stalls reduce performance, but sometimes inevitable
- To avoid nops a programmer or compiler can reorder code