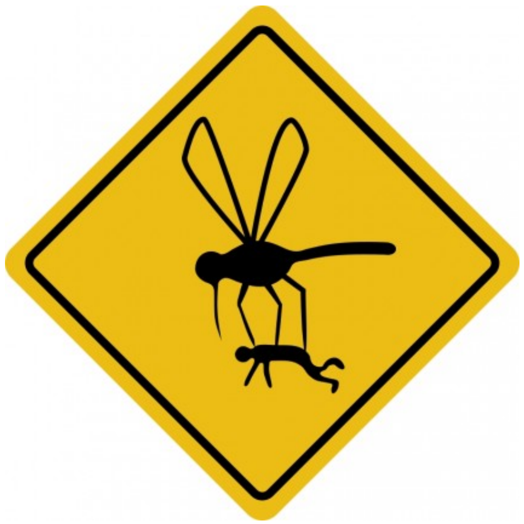
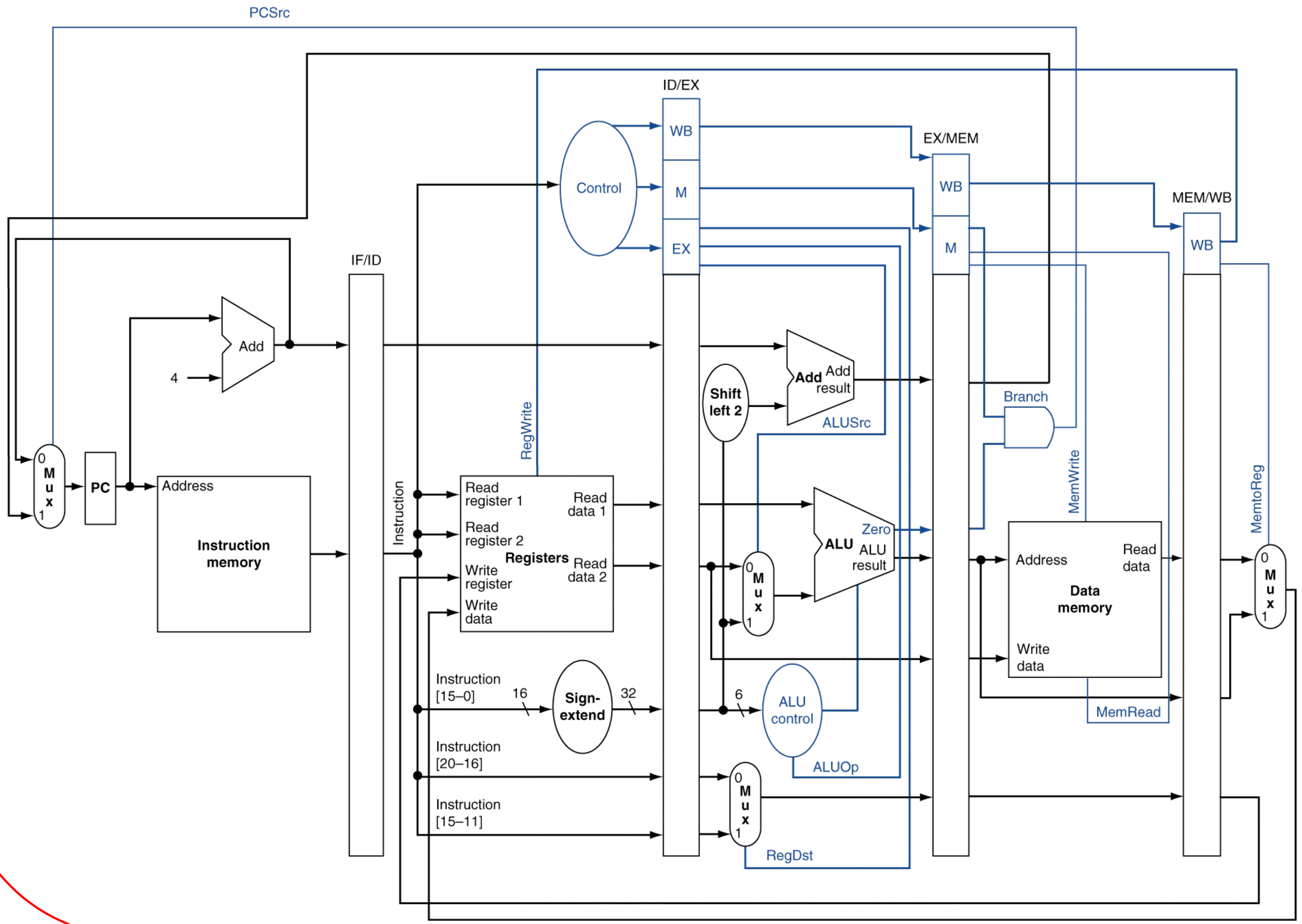
NWEN 242

Pipelining hazards / data hazards 1



Review: pipelined datapath with control signals



Agenda

- Hazards
  - Structural hazards
  - Data hazards
  - Control hazards
- Data hazards
  - Fixing hazards

Hazards

Situations that prevent starting
the next instruction in the next cycle

- Structural hazards
  - A required resource is busy
- Data hazards
  - Need to wait for previous instruction to complete its data read/write
- Control hazards
  - Deciding on control action depends on previous instruction

Structural hazards

- Conflicting demands for use of a resource

- In MIPS we use Instruction memory and Data memory
  - Two hardware resources dedicated to accessing Instructions and Data
  - What if only one resource was used?
    - Single cycle would cope ok
    - Pipelined datapaths will try using the same resource twice

- Hence, pipelined datapaths require separate hardware for accessing instruction/data memories

Data hazards

Where an instruction depends on the result of a previous instruction.

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

Control Hazards

Branch hazards

- The branch determines control flow
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Identifying a branch instruction ==> ID (Instruction Decode)
    - Next instruction is already in ====> IF (Instruction Fetch)

Data hazards

Consider this sequence:

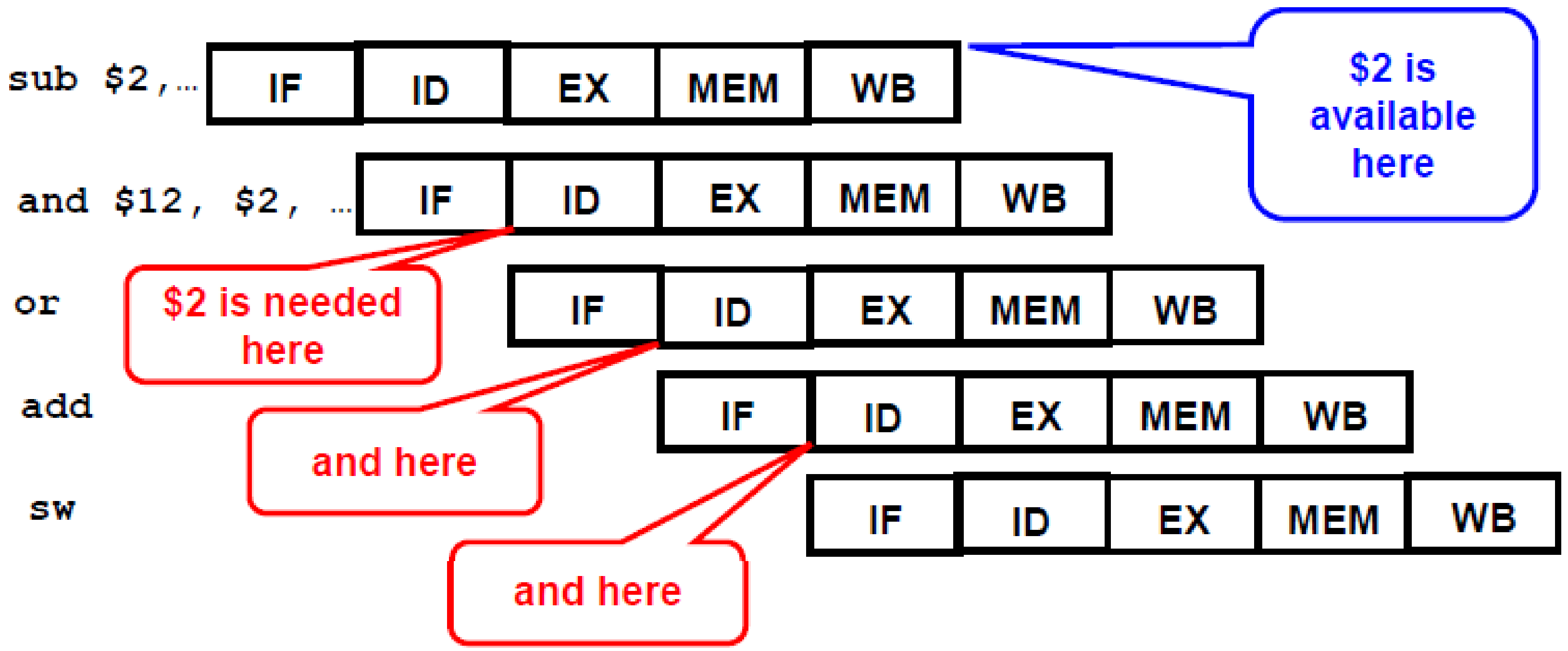
```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

Where are the data hazards?

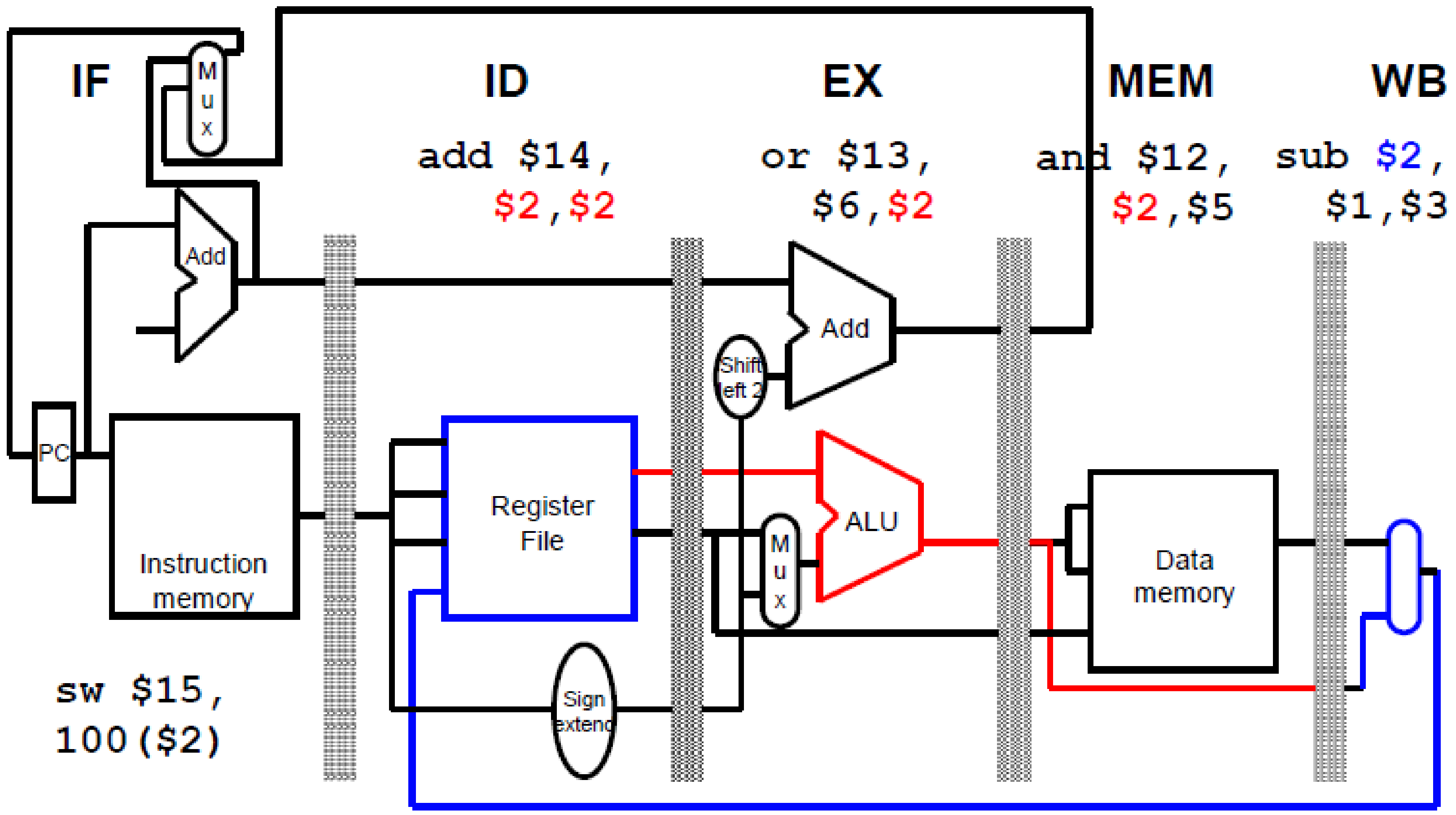
Data hazards in multiple-clock-cycle pipeline diagram

Recall:

- An R-type instruction writes ALU result into register file during WB (fifth) stage
- Register file is read in the ID (second) stage



Data hazards in single-clock-cycle pipeline diagram



Question for you

- Hazards make computation in a pipelined datapath unreliable
- The problem is aggravated if the Register File is read at the start and written at the end of a cycle
- What to do?
  a) Give up and abandon the pipelined datapath
  b) To consider hazard prevention techniques
  -> Bubbles/stalls/nops are sometimes inevitable

Insert bubbles / stalls / nops

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

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

nop	0	0	0	0	0	No Operation (do nothing)
	6	5	5	5	11	

Prevention of hazards

- Who should be responsible for detecting (and presumably correcting) them?
- It's probably not fair to ask the programmers to do this, because that means he/she needs to be knowledgeable about the intricacies of the pipeline
- So, the compiler or assembler can be programmed to know about the pipeline and do the detection

Compiler prevention of hazards

- Let's assume first that the compiler does the detection
  - It has a table telling it which instructions are potentially at risk.
  - This table says something like 'the result of an R-type instruction will not be available within 3 cycles, so make sure that those three cycles don't need to read from the register file'.
- Once the compiler has detected a hazard it can fix the hazard by inserting nop instructions. "nop" stands for 'no operation'
  - It can be a real operation like or \$0, \$0, \$0, which does nothing, or
  - A pseudo instruction just like nop
  - nops act like bubbles

Compiler prevention of data hazards

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

→

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

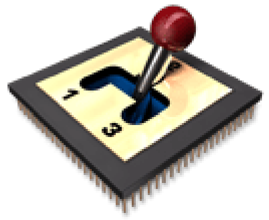
nop

0	0	0	0	0
6	5	5	5	11

No Operation (do nothing)

using nops == 3 cycle bubble

Hardware prevention of data hazards



- Hardware detects hazard in AND instruction:
  - AND is stalled
  - bubbles (nops) are inserted in the pipeline until hazard goes away
  - bubbles move through pipeline just as ordinary instructions

- Detection occurs in ID stage
  - Instruction wants to read a register to be written by an instruction in the EX, MEM, or WB stages
  - Hazard clears when conflicting instruction leaves WB stage

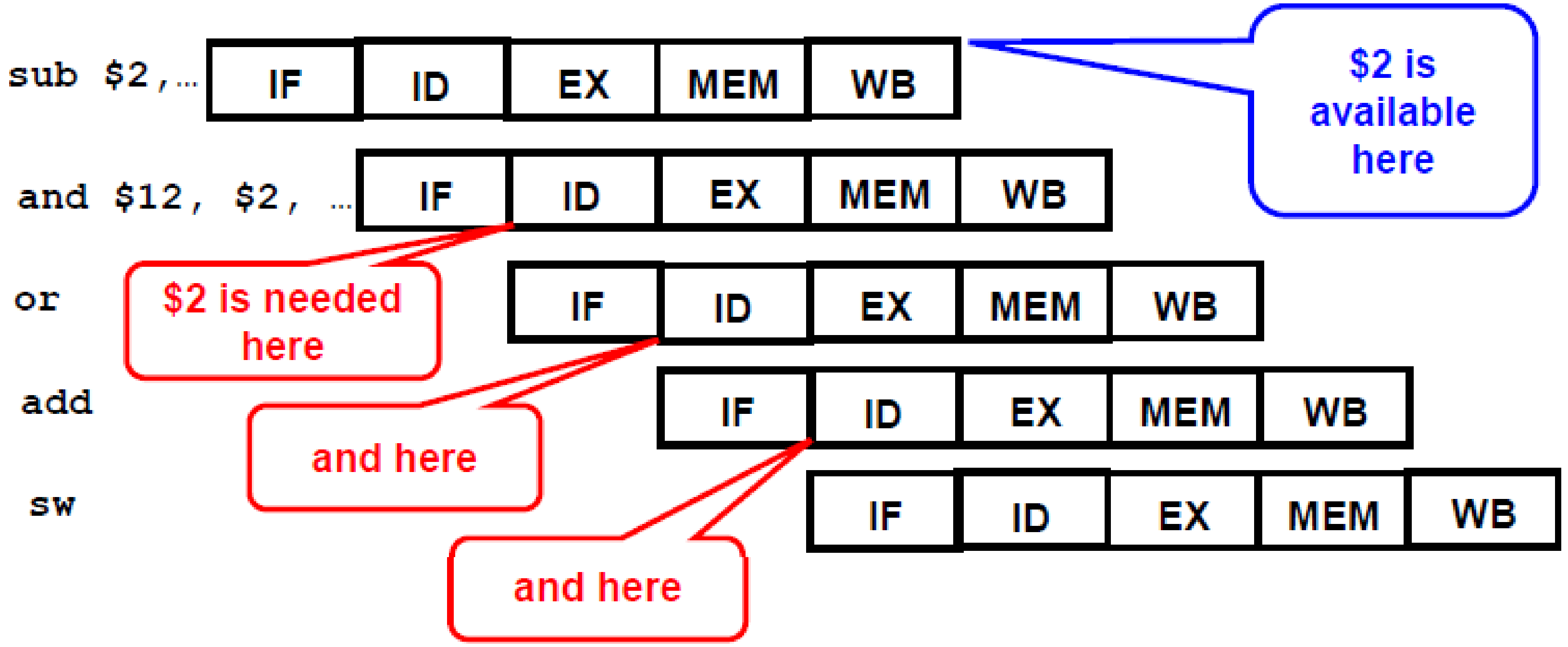
```
sub   $2,   $1,   $3
and   $12,  $2,   $5
or    $13,  $6,   $2
add   $14,  $2,   $2
sw    $15,  100 ($2)
```

R/W behavior of register file

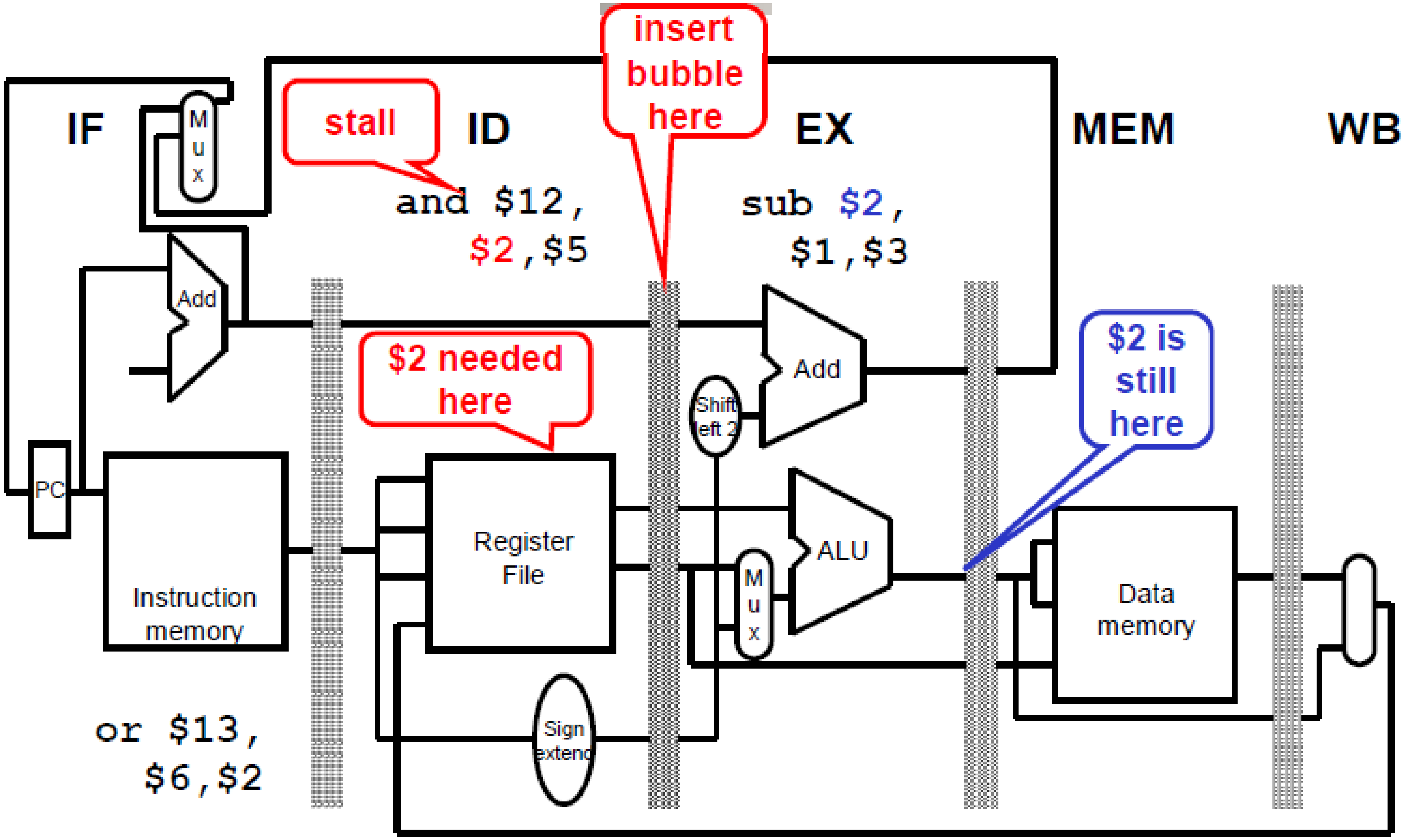
- From now on, when considering hardware detection and prevention of hazards, we shall assume that our pipelined datapath contains a special register file
- Register file is special in the sense that it is written in the first half, and read in the second half of a processor clock
- The consequence is that data written by an instruction being in the WB stage, will be read by an instruction being in the ID stage during the same clock cycle
- So, only two bubbles will be needed between two consecutive R-type instructions

Data hazards in multiple-clock-cycle pipeline diagram

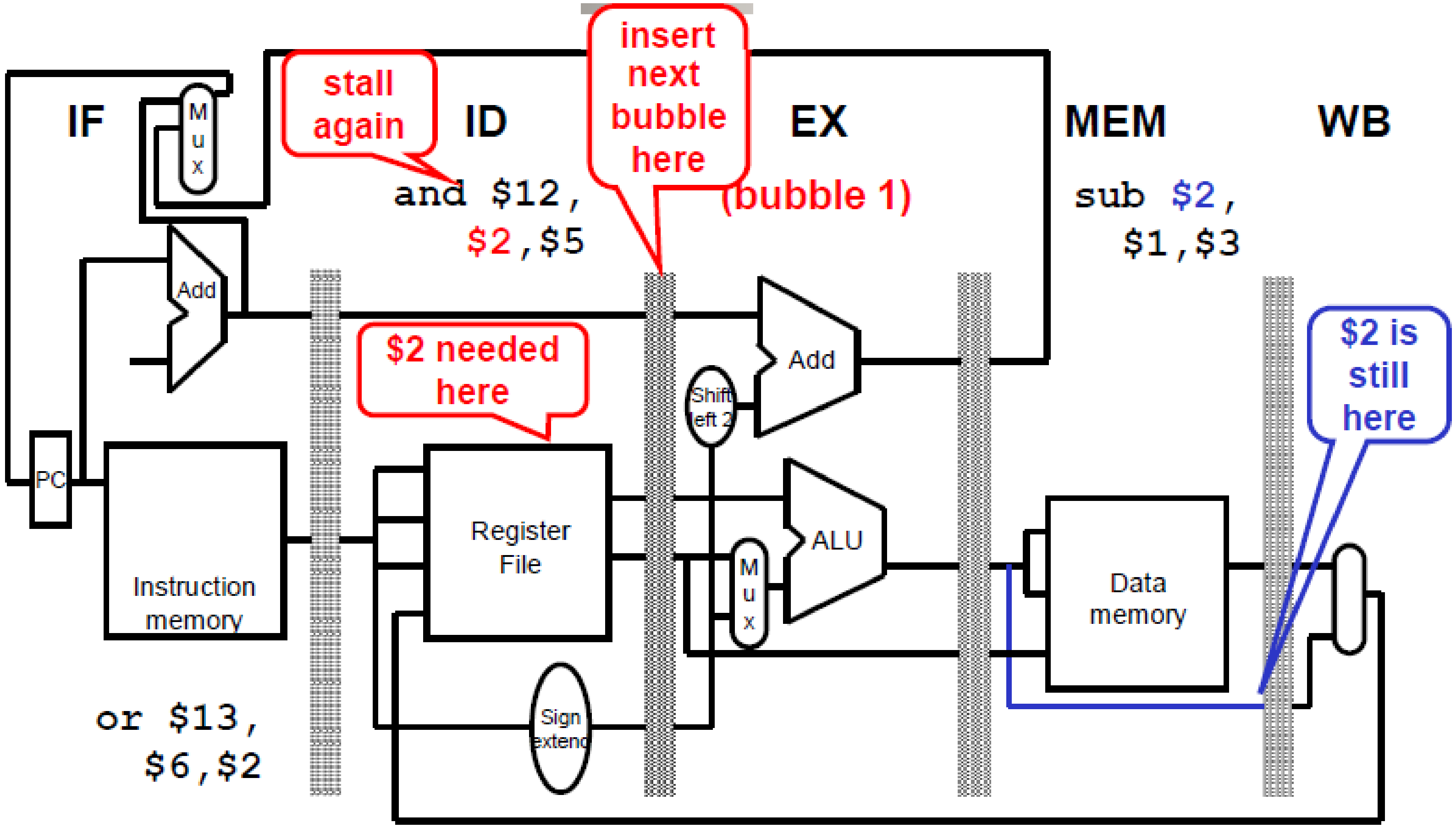
- Recall:
  - An R-type instruction writes ALU result into register file during WB (fifth) stage
  - Register file is read in the ID (second) stage



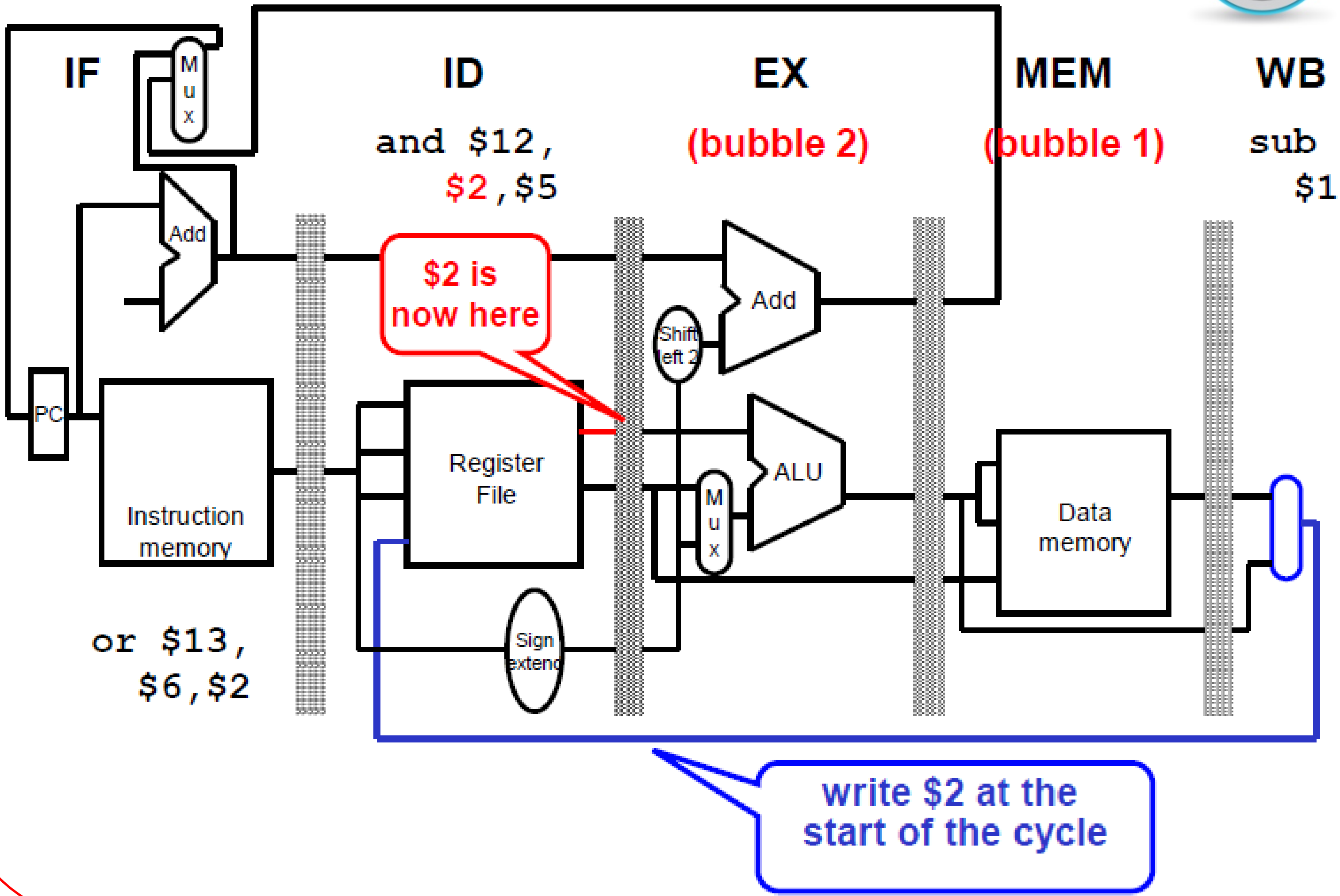
Stalled pipeline: insert 1st bubble



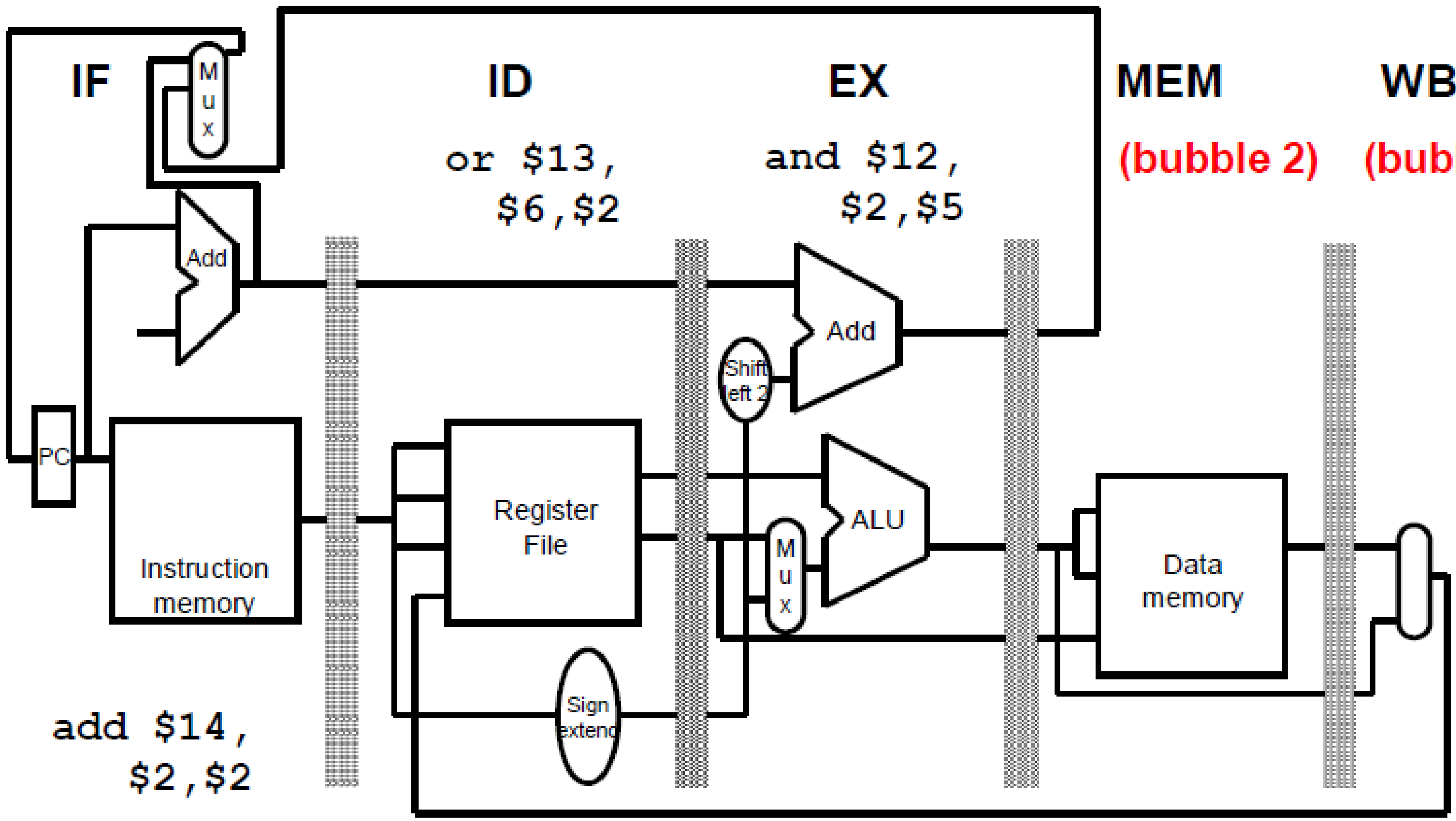
Stalled pipeline: insert 2nd bubble



Pipeline is fully active again



Pipeline after the hazard



How to create a nop / bubble?

- ID stage detects a hazard
- PC is prevented from being updated
  - So IF stage will fetch the same instruction again
  - No new instructions can enter the pipeline
- IF/ID register is prevented from being updated
  - So ID stage will decode the same instruction again
- The EX, MEM and WB control fields of ID/EX register is zero-ed to create the bubble
  - Control values are all 0
  - No registers or memories are written
- We keep stalling until the hazard has gone

Stalling a pipeline

- A simple solution
- Performance penalty
- There are cases that you have to stall
- You may want to avoid stalling when you can
- Forwarding

using stalling == 2 cycle bubble

Forwarding

- *Forwarding* the data as soon as it is available to any units that need it before it is available to read from 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

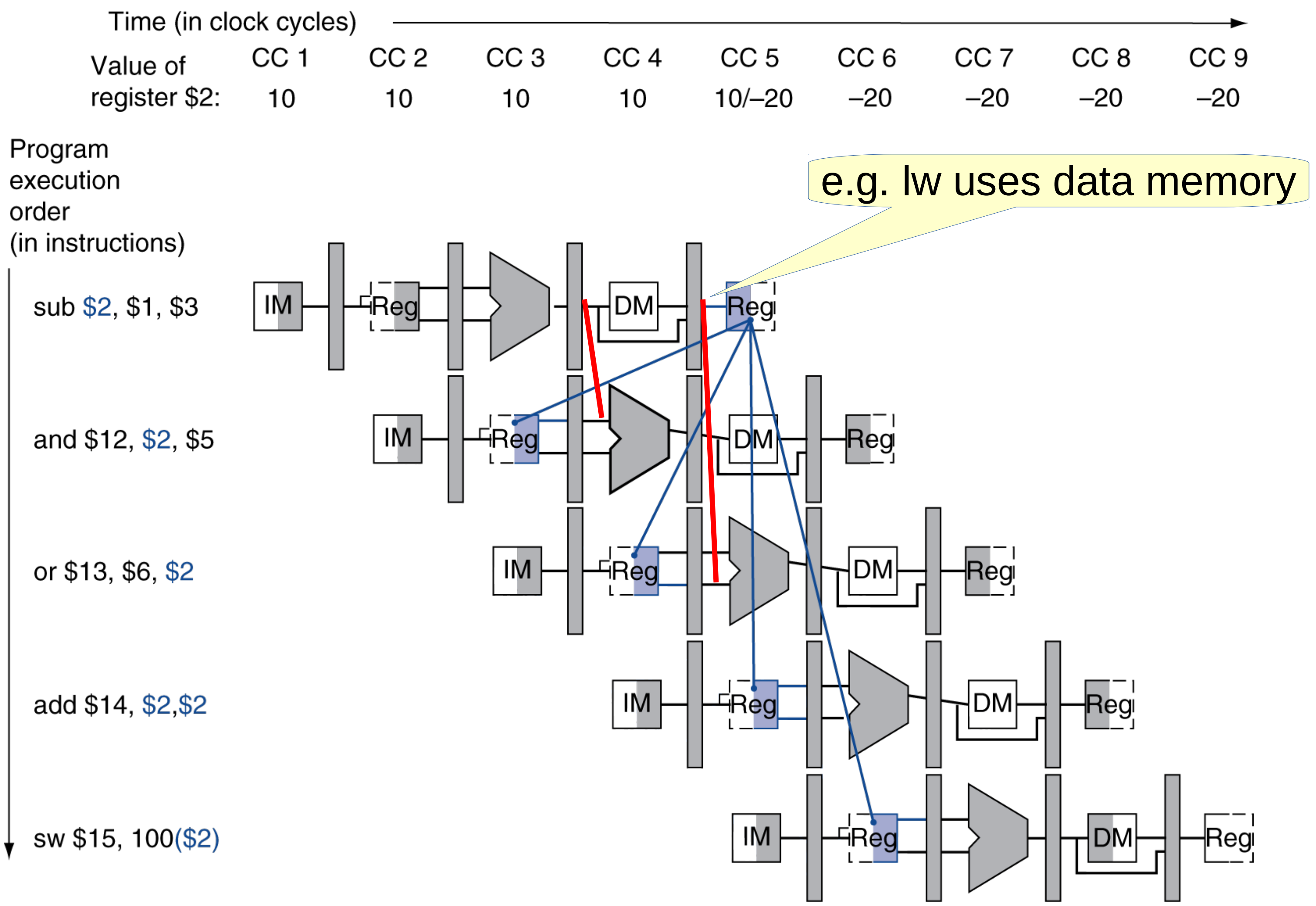
Data Hazards

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

using forwarding == 0 cycle bubble between two R-types

using forwarding == 1 cycle bubble between lw and R-type

Summary

- Three types of hazards:
  - Structural hazards
  - Data hazards
  - Control hazards
- Register file is written in the first half, and read in the second half of a clock cycle
- The use of nops / bubbles to prevent data hazards
  - Compiler (software) / hardware solutions
- Stalls/nops cause performance penalty
- Forwarding