ECE260: Fundamentals of Computer Engineering

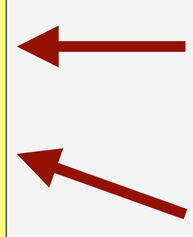
Data Hazards in a Pipelined Datapath

James Moscola
Dept. of Engineering & Computer Science
York College of Pennsylvania

Data Hazards in ALU Instructions

- Next instruction needs to wait for previous instruction to complete its data read/write
  - A data dependency exists between the result of one instruction and the next
  - Extremely common!
- Consider the following sequence:

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



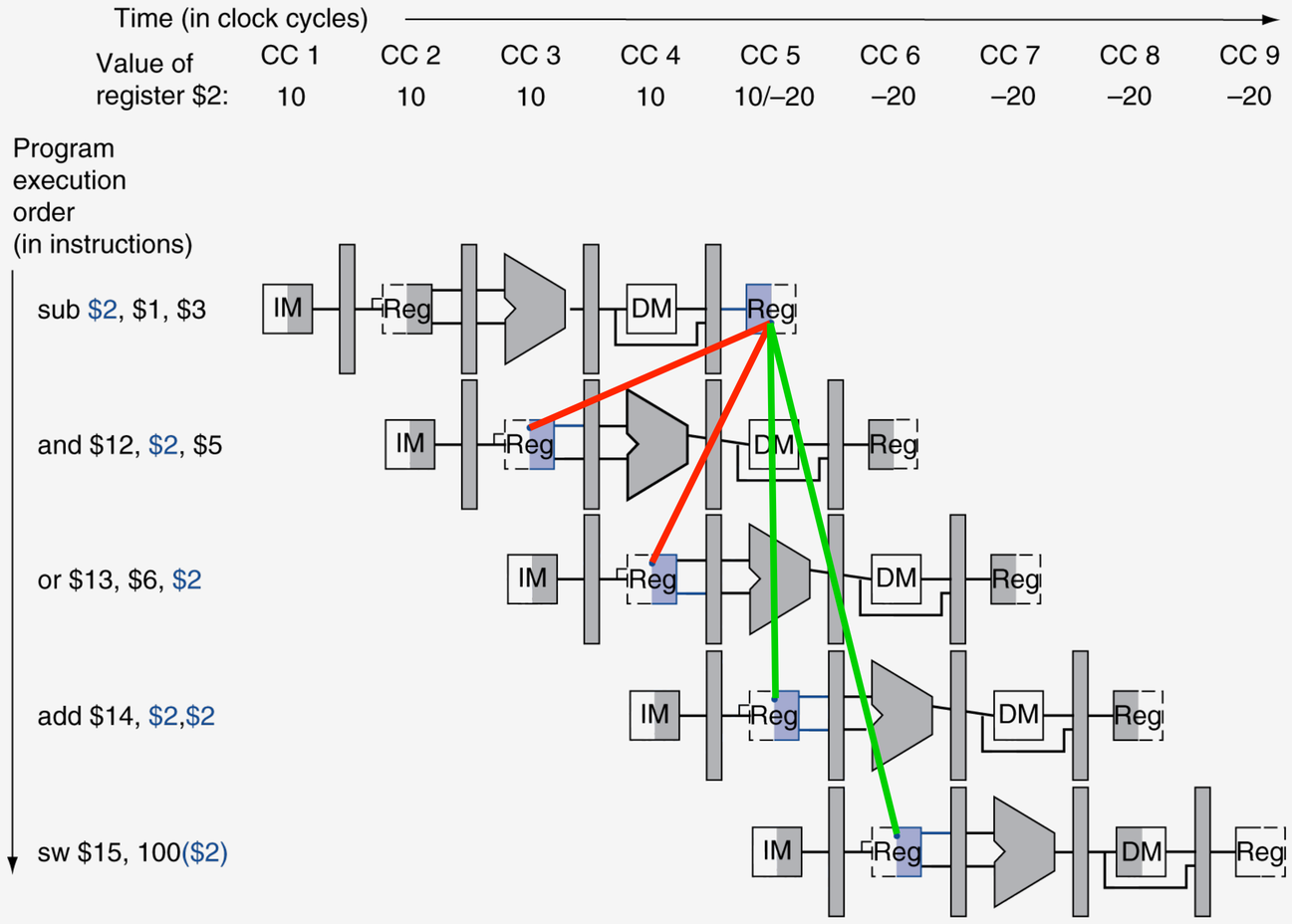
the and instruction requires $2 as input
the sub instruction will not yet have executed WB stage to write result
Register file will contain "stale" data for $2 when and requests it

the or instruction has a similar issue

- Data hazards can be resolved with forwarding
  - Need a way to determine when forwarding should happen
  - Need hardware support to enable forwarding

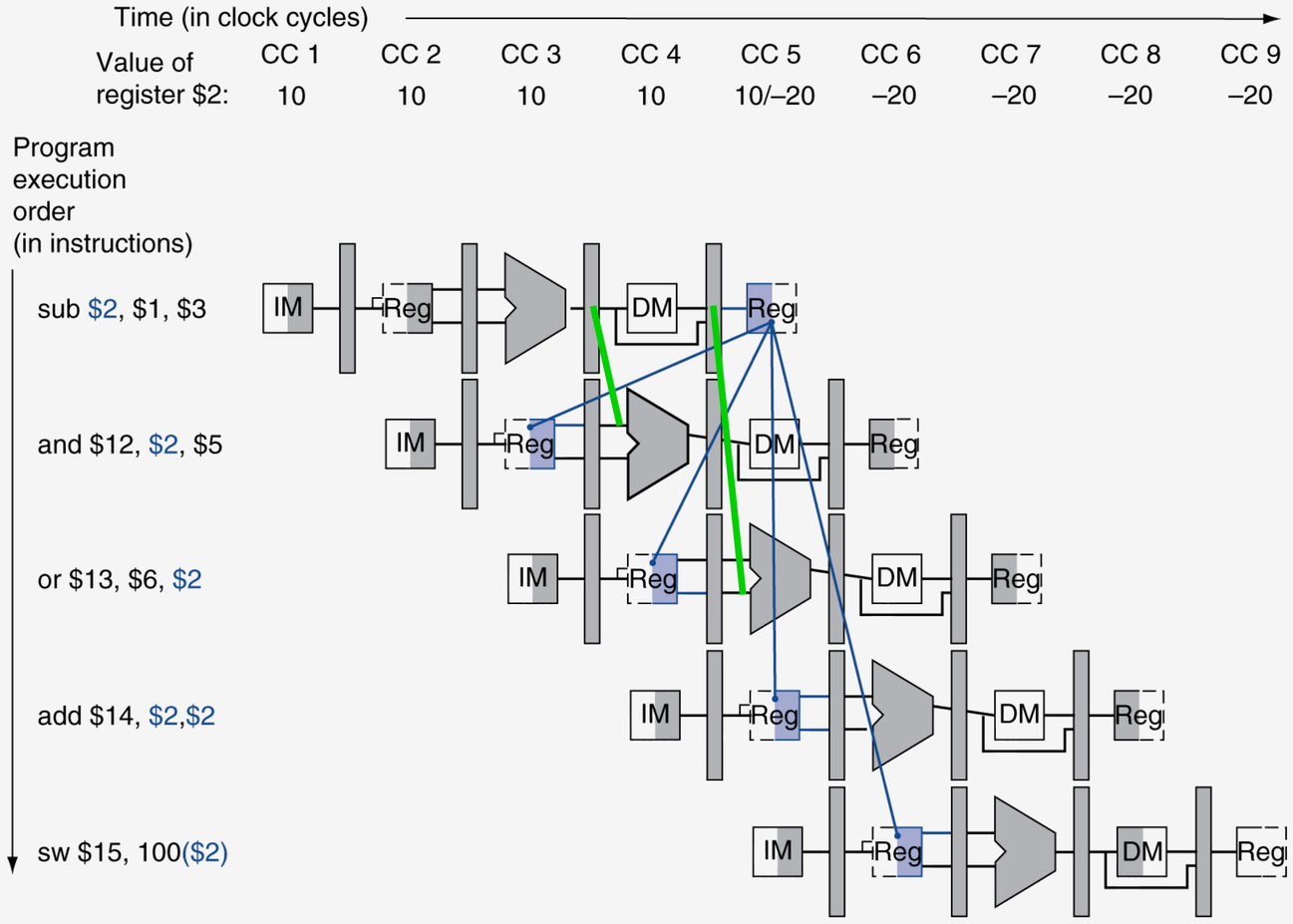
Data Hazards in ALU Instructions (continued)

- A multi-cycle pipeline diagram illustrates the data dependencies
  - Red lines in the diagram cannot be satisfied without pipeline stalls or forwarding support
  - Green lines require no extra hardware support or pipeline stalls



Dependencies & Forwarding

- Desired data is available at end of EX stage during clock cycle 3
- Desired behavior is shown as green lines on the multi-cycle pipeline diagram
  - Forward data from output registers of EX and MEM stages to input of ALU
  - Forward data to ***either*** input of ALU (maybe even both)
- No need to forward from WB stage



Detecting the Need to Forward

- Must pass register numbers along pipeline and store in pipeline registers
  - Register numbers for Rs and Rt are passed from ID stage to EX stage
  - Register number for Rd is passed through entire pipeline
    - Were already passing this value down pipeline anyway to store result of instruction in register file
    - In MEM and WB, the "Rd" field may have come from the Rt field of the instruction (recall that the I-Type instructions use the Rt field as the destination register!)
- New notation is used to identify register numbers at different stages of pipeline
  - **ID/EX.RegisterRs** = the register # for the Rs register that is stored in the ID/EX register
  - **ID/EX.RegisterRt** = the register # for the Rt register that is stored in the ID/EX register
  - **EX/MEM.RegisterRd** = the register # for the Rd register that is stored in the EX/MEM register
  - **MEM/WB.RegisterRd** = the register # for the Rd register that is stored in the MEM/WB register

Detecting the Need to Forward (continued)

- Data hazards ***might exist*** when:
  - **1a.** EX/MEM.RegisterRd == ID/EX.RegisterRs
    - Maybe forward data from EX/MEM pipeline register to **1st** input of ALU
  - **1b.** EX/MEM.RegisterRd == ID/EX.RegisterRt
    - Maybe forward data from EX/MEM pipeline register to **2nd** input of ALU
  - **2a.** MEM/WB.RegisterRd == ID/EX.RegisterRs
    - Maybe forward data from MEM/WB pipeline register to **1st** input of ALU
  - **2b.** MEM/WB.RegisterRd == ID/EX.RegisterRt
    - Maybe forward data from MEM/WB pipeline register to **2nd** input of ALU
- Not all instructions write the Rd register, so forwarding may not be necessary
  - Example: the **sw** instruction does not write the register file

Detecting the Need to Forward (continued)

- **Only forward data if the forwarding instruction will write to the register file**
  - Check the control signals that are in the pipeline to determine if instruction will write to register file
    - If the RegWrite control signal is asserted for an instruction:
      - it will write the register file during the WB stage
      - forward data from EX/MEM or MEM/WB as needed
- **Do NOT forward data if the destination register Rd is register $0!**
  - Recall that register $0 ($zero) is read-only
  - Setting $0 as a destination register will not actually write the register file

Forwarding Paths

NOTE: Forwarding unit can pass EX/MEM.RegisterRd to either input of the ALU

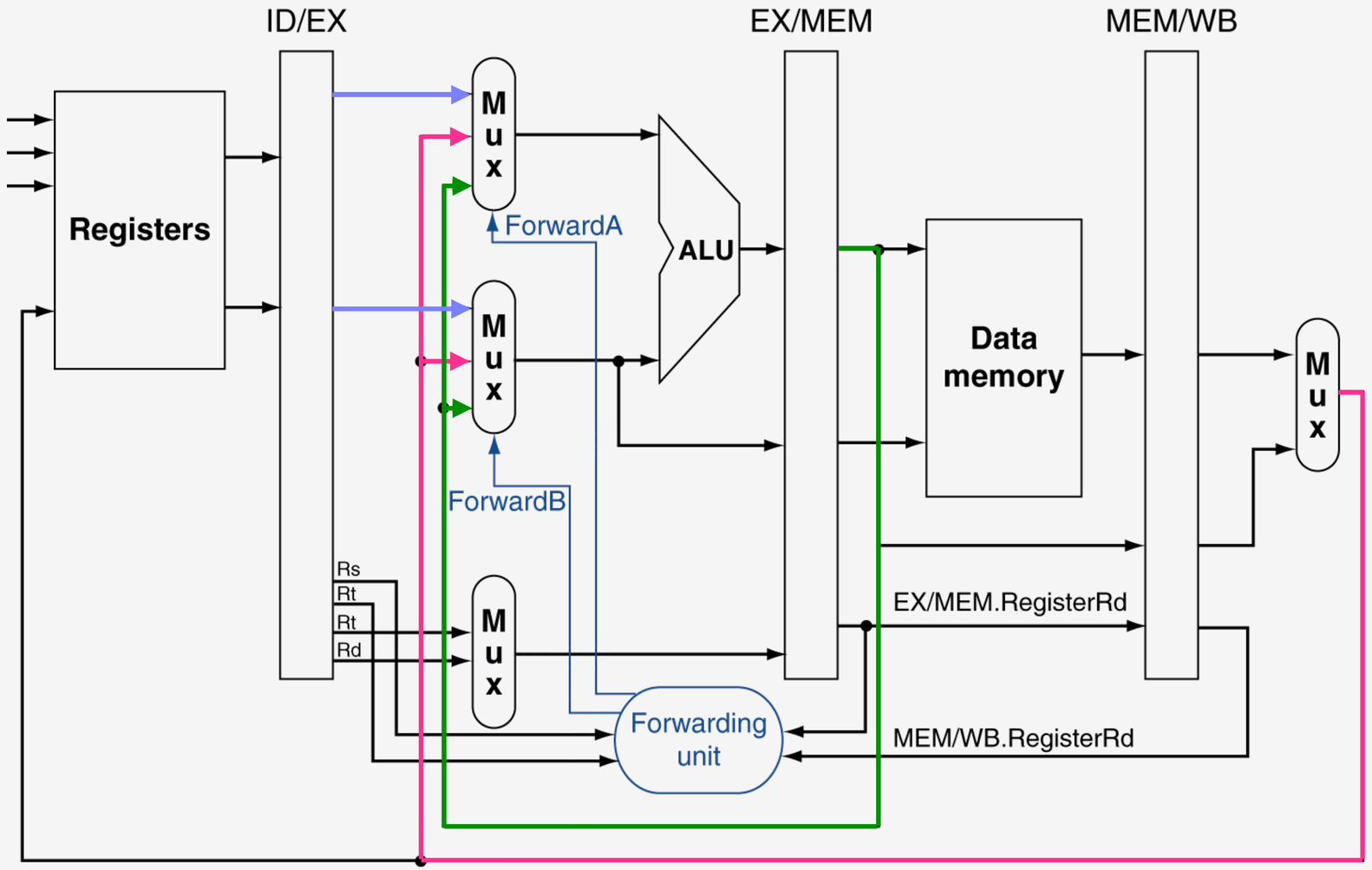
NOTE 2: Forwarding unit can pass MEM/WB.RegisterRd to either input of the ALU

NOTE 3: The EX/MEM.RegisterRd field can come from either ID/EX.RegisterRd **OR** from ID/EX.RegisterRt

NOTE 4: The Forwarding Unit compares the Rd field at the MEM and WB stages to the ALU input registers in the EX stage

Legend:

- no forwarding, value from register file
- value forwarded from EX/MEM
- value forwarded from MEM/WB



Forwarding Conditions

- EX hazard (→ from previous slide):
  - Forward EX/MEM.RegisterRd to ALU input 1

```
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
  ForwardA = 10two
```

  - Forward EX/MEM.RegisterRd to ALU input 2

```
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
  ForwardB = 10two
```

- MEM hazard (→ from previous slide):
  - Forward MEM/WB.RegisterRd to ALU input 1

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
  ForwardA = 01two
```

  - Forward MEM/WB.RegisterRd to ALU input 2

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
 ForwardB = 01two
```

Double Data Hazard (seriously?)

- Consider the sequence:

```
add $1, $1, $2
add $1, $1, $3   # Condition indicates forwarding data from EX/MEM
add $1, $1, $4   # Condition indicates forwarding data from EX/MEM AND from MEM/WB
```

- Both hazards occur when the third **add** instruction is in the EX stage
  - Forwarding conditions suggest that data should be forward from EX/MEM stage
  - Forwarding conditions **ALSO** suggest that data should be forward from MEM/WB stage
- Can't forward data to ALU input from two different places
  - Definitely want to use the most recent result (i.e. the result in the EX/MEM register)
- Revise MEM hazard condition, only forward if EX hazard condition isn't true
  - i.e. forwarding from EX/MEM takes precedence over forwarding from MEM/WB

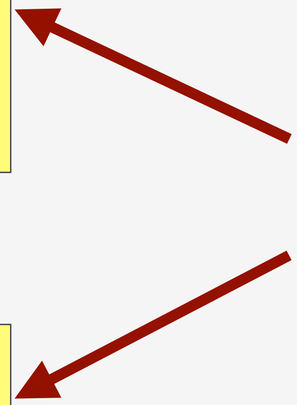
Revised Forwarding Conditions

- EX hazard: (same as before)
- MEM hazard:
  - Forward MEM/WB.RegisterRd to ALU input 1

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
  ForwardA = 01two
```

  - Forward MEM/WB.RegisterRd to ALU input 2

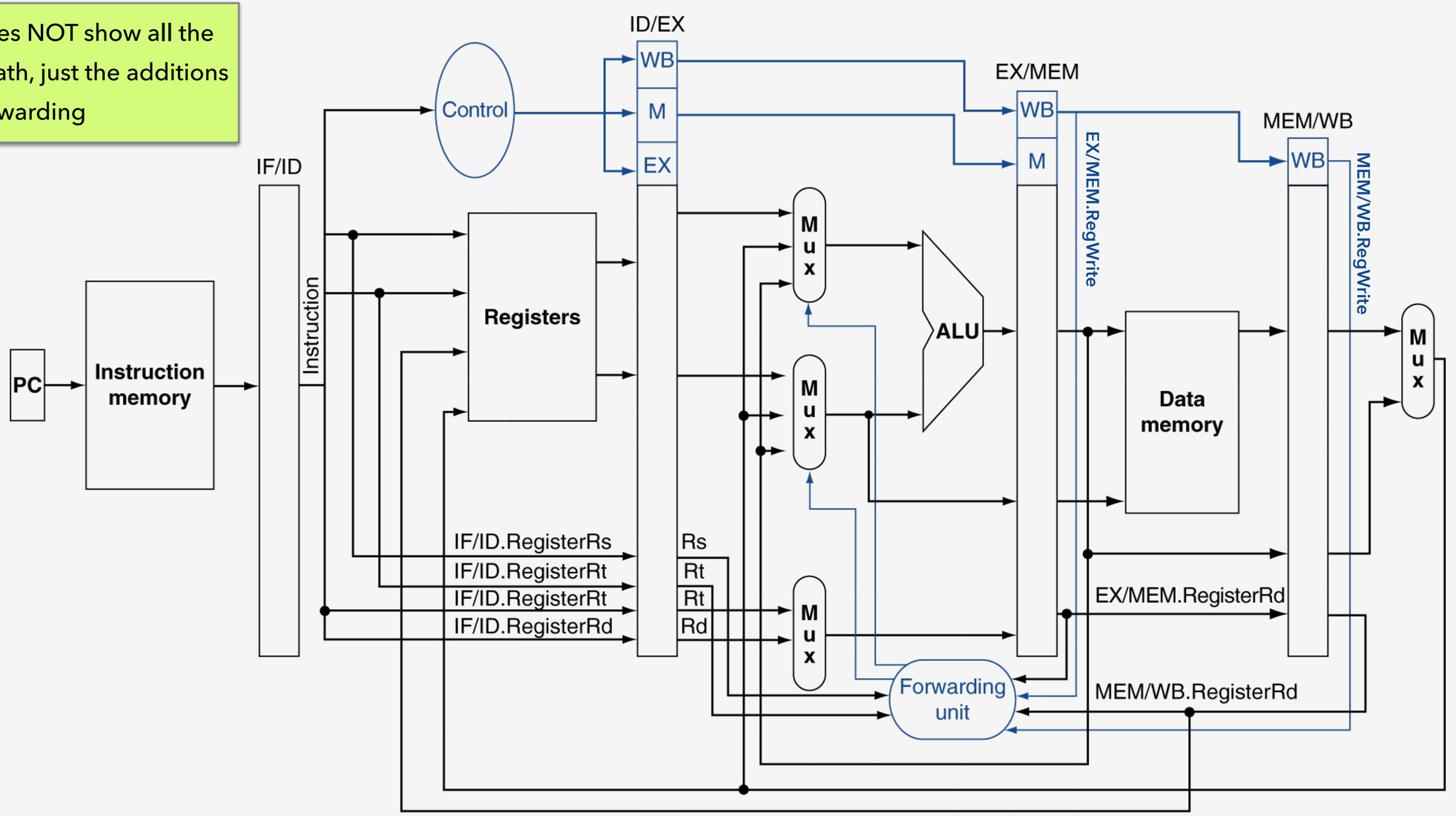
```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
  ForwardB = 01two
```



Addition to MEM forwarding rule is shown in blue
Additional basically says,
"And NOT forwarding from EX/MEM"

Datapath Showing Forwarding Components

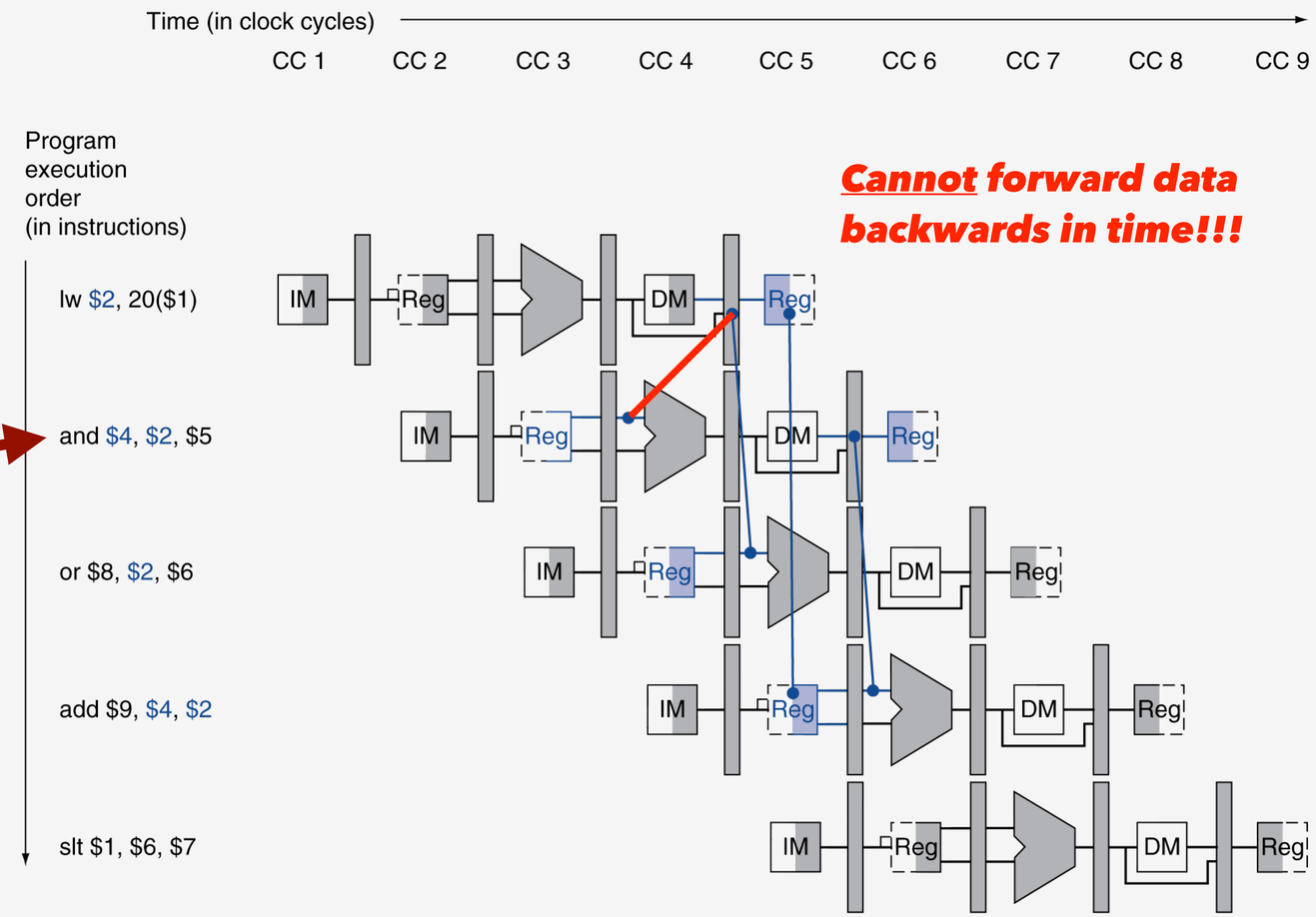
NOTE: This datapath does NOT show all the components of the datapath, just the additions made for forwarding



Load-Use Data Hazard

- Forwarding works very well in many cases, but not all
- Consider when a **lw** instruction precedes an instruction that uses the value being loaded
  - Loaded data is not available in MEM/WB until **AFTER** the next instruction needs it
  - Need to stall for one clock cycle
- Example code:

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```



Detecting a Load-Use Hazard

- Load-use hazard occurs under following condition:
  - Instruction is a load instruction as indicated by the MemRead control signal
  - Next instruction specifies a source operand register that is being written by the load instruction

```
ID/EX.MemRead and
  ((ID/EX.RegisterRt == IF/ID.RegisterRs) or
   (ID/EX.RegisterRt == IF/ID.RegisterRt))
```

- Check for hazard when the “using” (i.e. dependent) instruction is decoded in the ID stage
  - Is there a load in the EX stage that is loading a source operand register?
- If load-use hazard is detected, stall the pipeline and insert a bubble

How to Stall the Pipeline

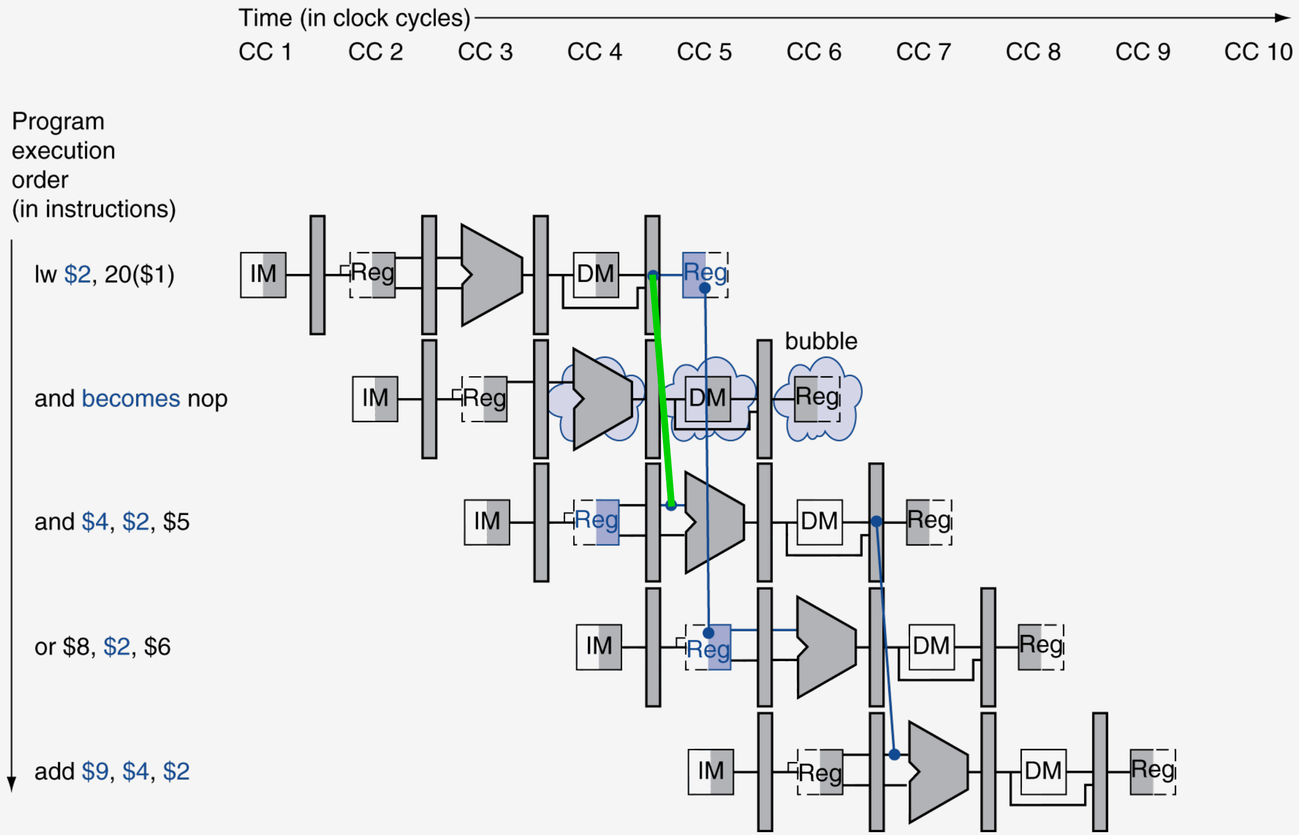
- Force control line values in ID/EX register to 0
  - Effectively results in a **nop** (no-operation) in the EX, MEM and WB stages (i.e. a bubble)
- Prevent update of PC and IF/ID register
  - Contents of PC and IF/ID will not change for the clock cycle that the pipeline is stalled
  - "Using" instruction is decoded a second time since it still exists in the IF/ID register
  - Instruction after the "using" instruction is fetched again (a second time)
- A 1-cycle stall allows MEM stage to read data for the **lw** instruction
  - Can subsequently forward data to the EX stage for the "using" instruction

Stalling the Pipeline (Inserting a Bubble)

- Example code:

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```

- Load-use hazard detected at clock cycle 3 when **and** instruction is decoded
  - Rs register for **and** instruction requires result of previous **lw** instruction
- Stall pipeline at clock cycle 4 to insert **nop**
  - Loaded data can then be forwarded to **and** instruction during CC5 (green line)

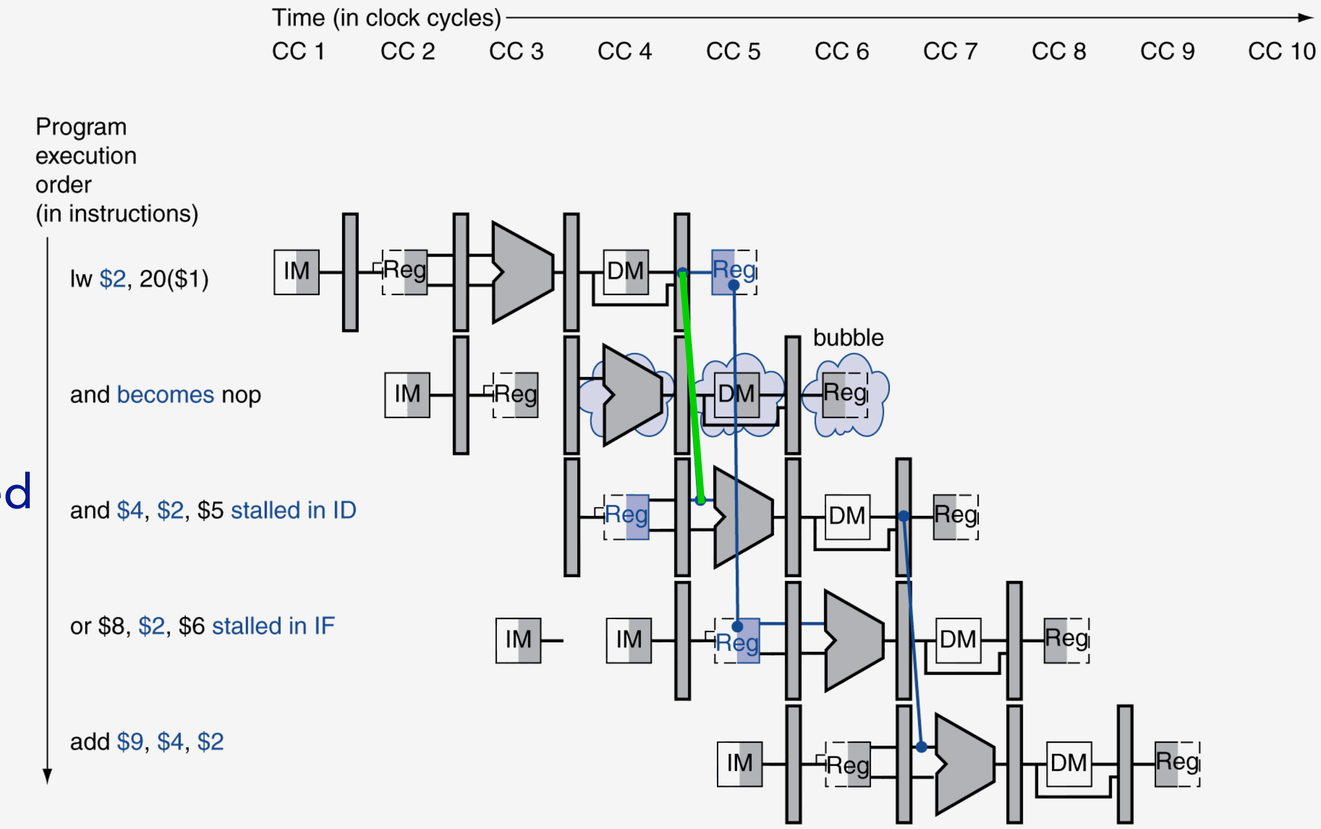


Stalling the Pipeline (Inserting a Bubble) (continued)

- Example code:

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```

- Slightly more accurate depiction shows:
  - **and** instruction is fetched at CC2, decoded at CC3, then decoded AGAIN in CC4
  - **or** instruction is fetched in CC3 and then fetched AGAIN in CC4



Datapath with Hazard Detection

NOTE: Hazard detection unit detects load-use hazards

NOTE: PCWrite is used to prevent PC from being written when pipeline is stalled

NOTE: IF/IDWrite is used to prevent PC from being written when pipeline is stalled