# CprE 381: Computer Organization and Assembly Level Programming

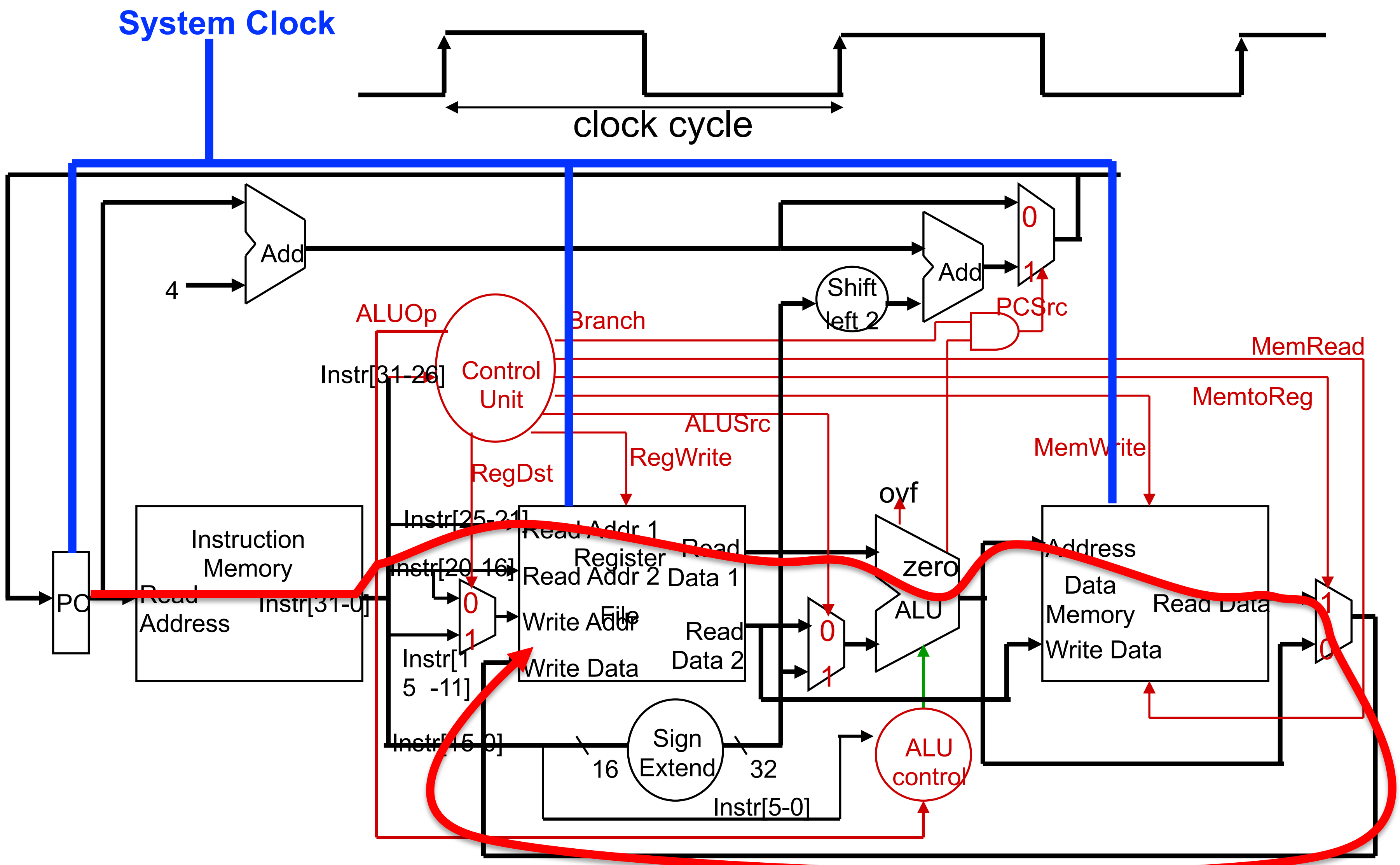
Data Hazards

Henry Duwe
Electrical and Computer Engineering
Iowa State University

## Administrative

- Term Proj 2 due this week in lab
  - For keeps
- HW7
  - Due Mar 27 11:59pm
  - Last HW for exam 2 coverage
  - Question 3 best exam prep if done well
- Exam 2
  - When: April 1
  - Where: **TBD**
  - What: MIPS Arithmetic through pipelining (including hazards)

# Review: Worst Case Timing (Load Instruction)



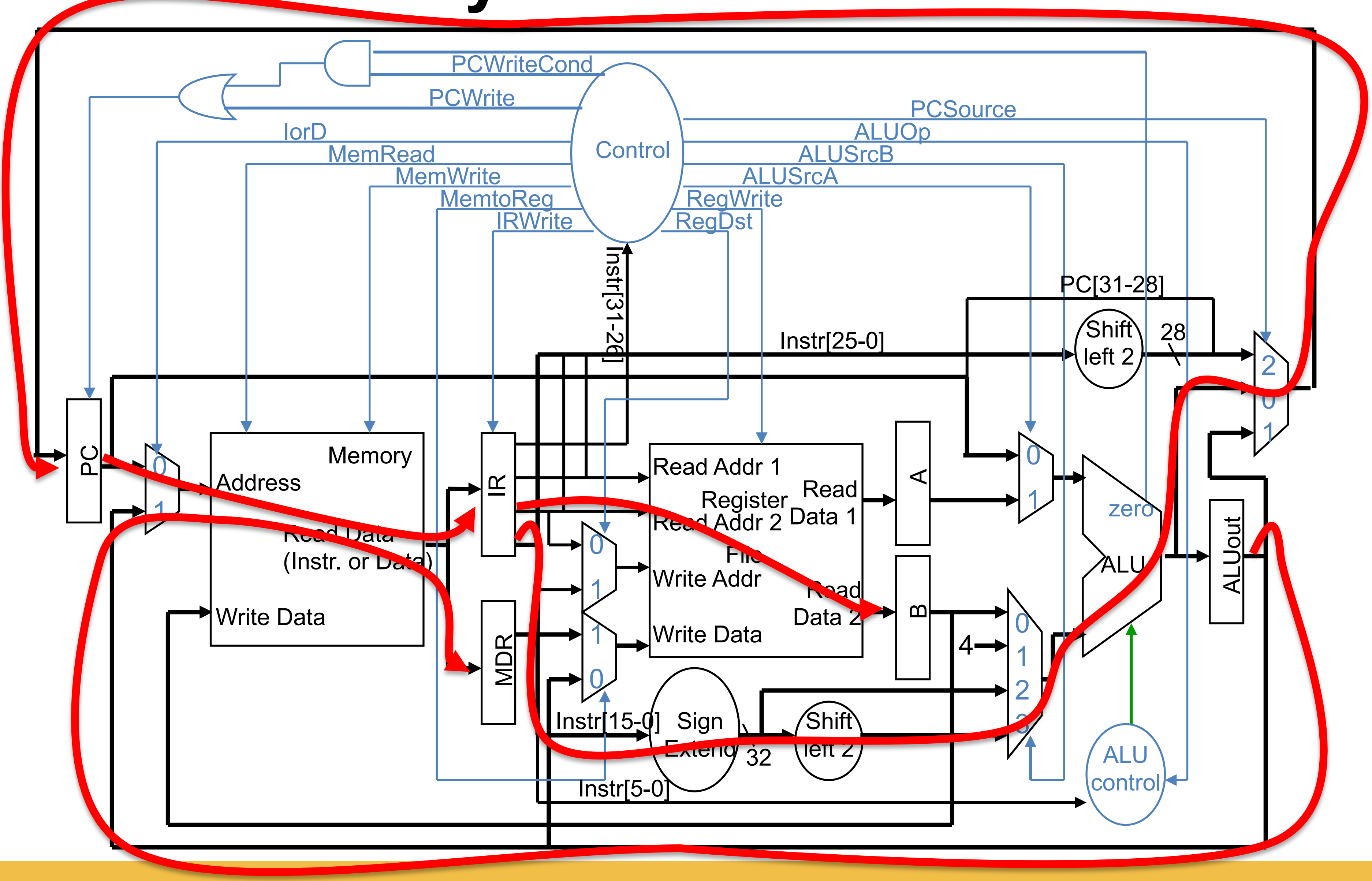
# Review: Execution Time

- Drawing on the previous equation:

$$\boldsymbol{Execution\,Time = \#\,Instructions \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}}$$

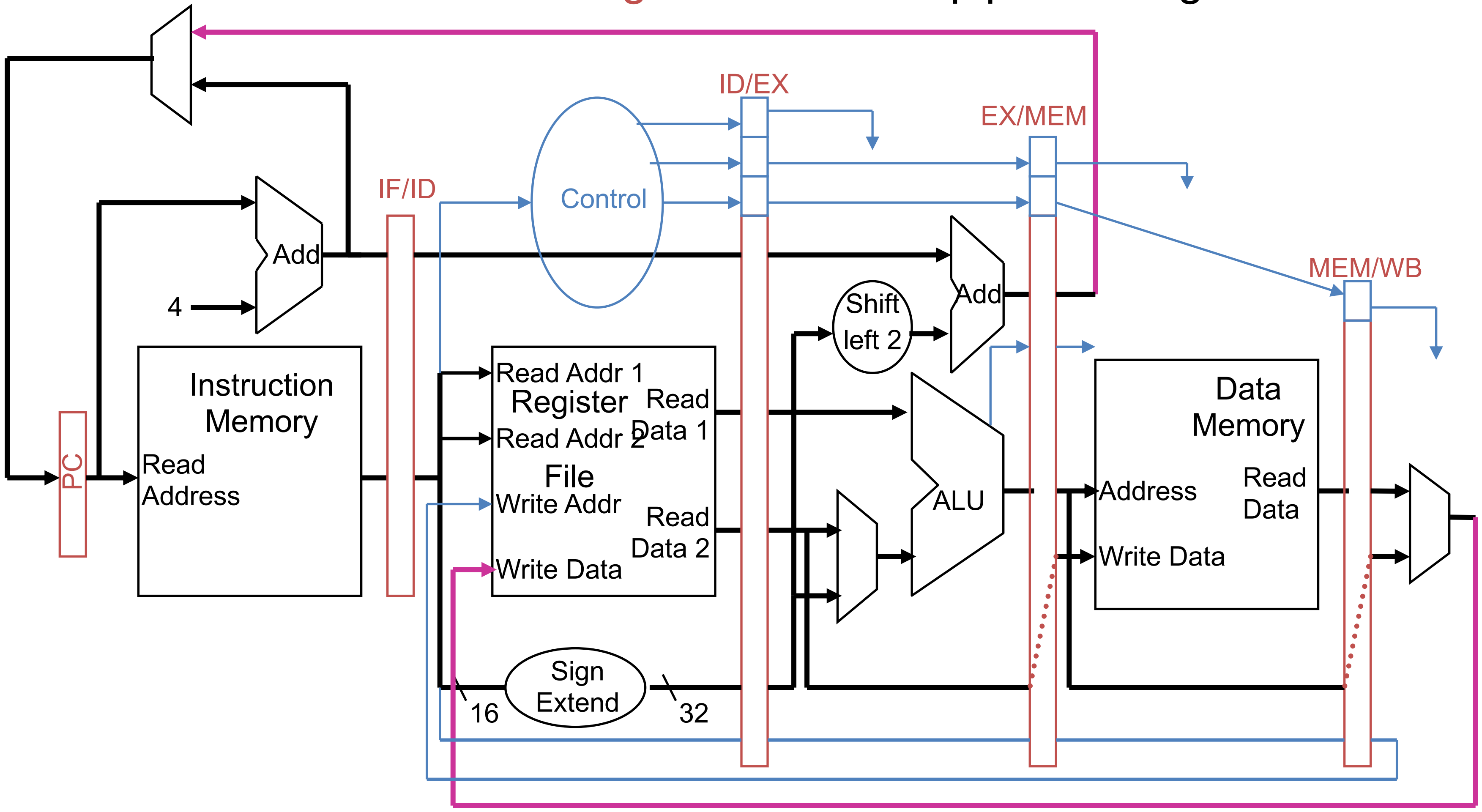
- To improve performance (i.e., reduce execution time)
  - Increase clock rate (decrease clock cycle time) OR
  - Decrease CPI OR
  - Reduce the number of instructions
- Designers balance cycle time against the number of cycles required
  - Improving one factor may make the other one worse...

# Review: Multicycle Processor



# Review: A Simple MIPS Pipeline

- All control signals can be determined during Decode
  - And held in the state registers between pipeline stages



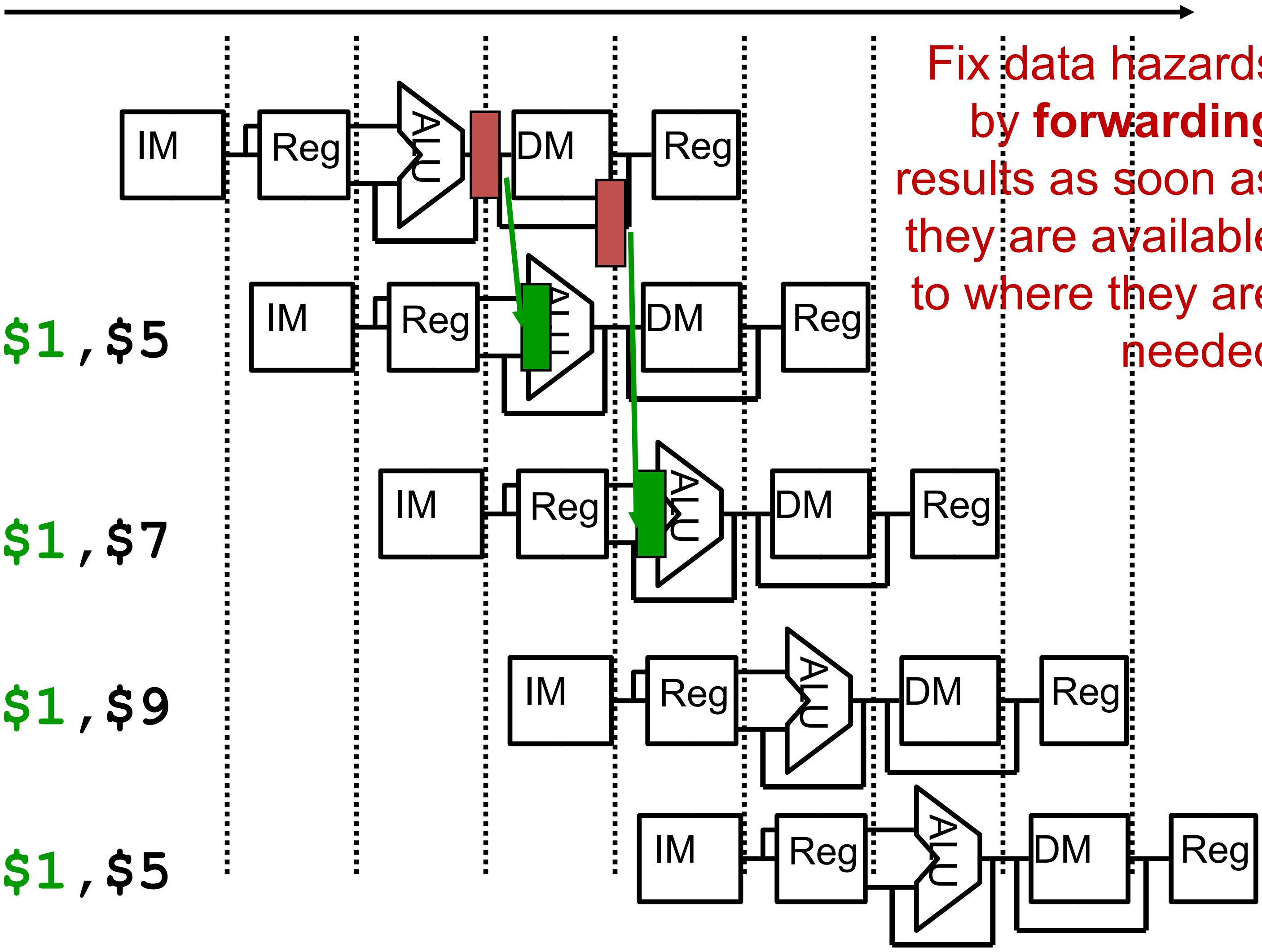
# Review: Oh, the *Hazards* I've Seen

- Pipeline Hazards
  - Structural hazards: attempt to use the same resource by two different instructions at the same time
  - Data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - Control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - Branch and jump instructions, exceptions
- Can always resolve hazards by **waiting**
  - Pipeline control must detect the hazard
  - And take action to resolve hazards

# Review: Forwarding to “Fix” a Data Hazard

Instr. Order

```
add $1,
sub $4,$1,$5
and $6,$1,$7
or  $8,$1,$9
xor $4,$1,$5
```



Fix data hazards by **forwarding** results as soon as they are available to where they are needed

# Implementing Forwarding Paths

- **WARNING**: DETAILED, METHODICAL WORK NEEDED

- Simple procedure → but need to cover all cases
  - Identify all pipeline stages that produce new values
    - In our case, EX and MEM
  - All pipeline registers after the earliest producer can be the source of a forwarded operand
    - In our case, EX/MEM, MEM/WB (although this is multiple registers)
  - Identify all pipeline stages that really consume values
    - In our case, EX (both A and B ports) and MEM (only data port)
  - These stages are the destinations of a forwarded operand
  - Add multiplexor input for each pair of source/destination stages

# Detecting Data Hazards

- 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
- *Possible* Data hazards when

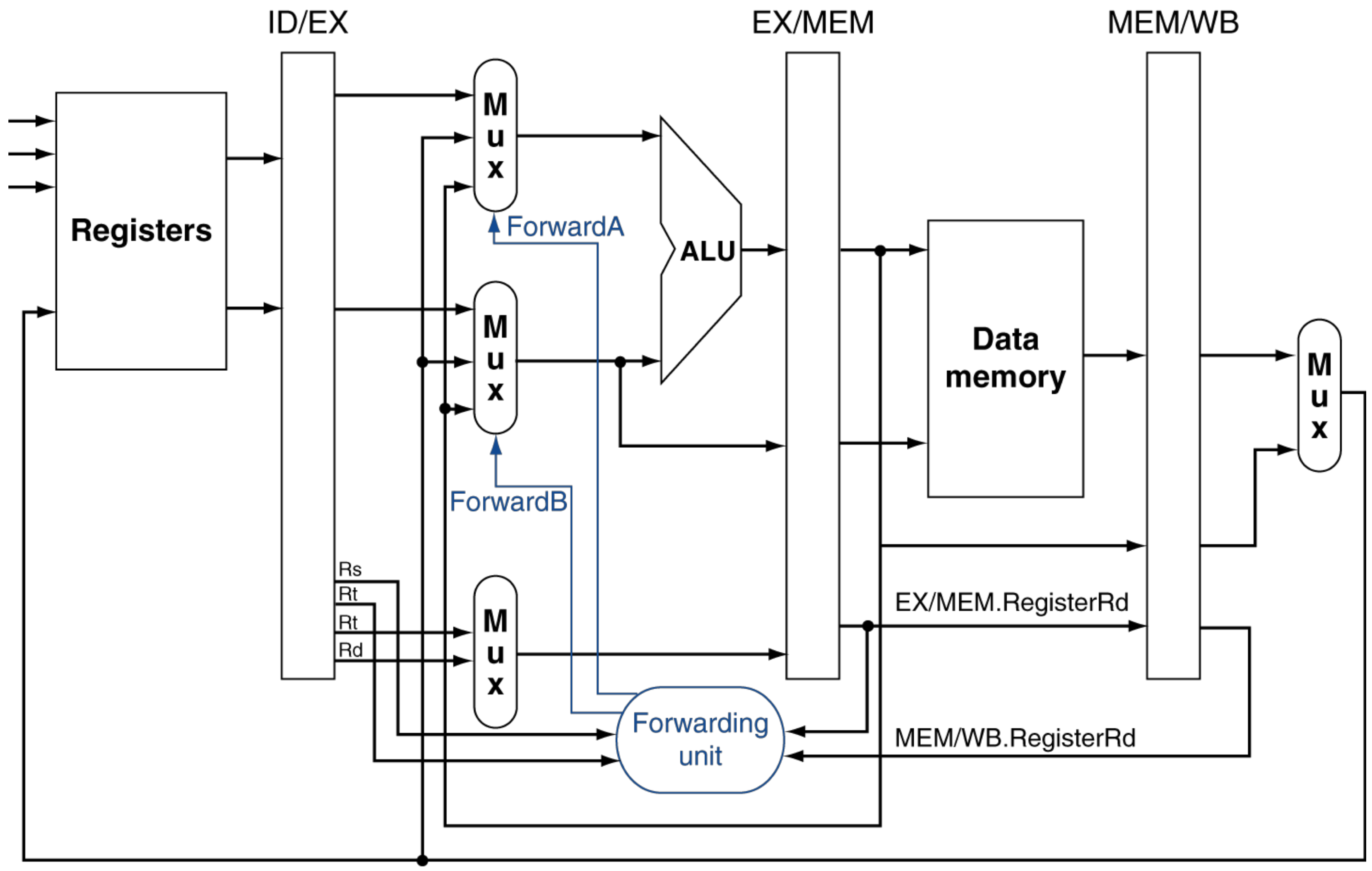
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
} Fwd from EX/MEM pipeline reg

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
} Fwd from MEM/WB pipeline reg

## Determining Forwarding Paths

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- MIPS Specific: And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

# Forwarding Hardware (ALU Src)



b. With forwarding

# Forwarding Hardware – Control

- Need to decide which multiplexer input to enable
  - Doesn't seem that hard but it can get troublesome

- How to implement the control when instruction j in ID stage
  - For each input operand of the instruction
    - Check if j actually reads that input operand
    - Check if instruction i in EX or MEM stage writes to same register
    - If there are multiple matches, pick **youngest** result (EX)
      - Set multiplexer control to select proper forwarding value
    - If instruction i is a load, and j is an ALU then stall instead

- If the instruction stalls
  - Control is re-evaluated in the next cycle
  - Until the instruction can move forward

- Watch out for naming issues in your VHDL implementation

# Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

## Double Data Hazard

- Consider the sequence:

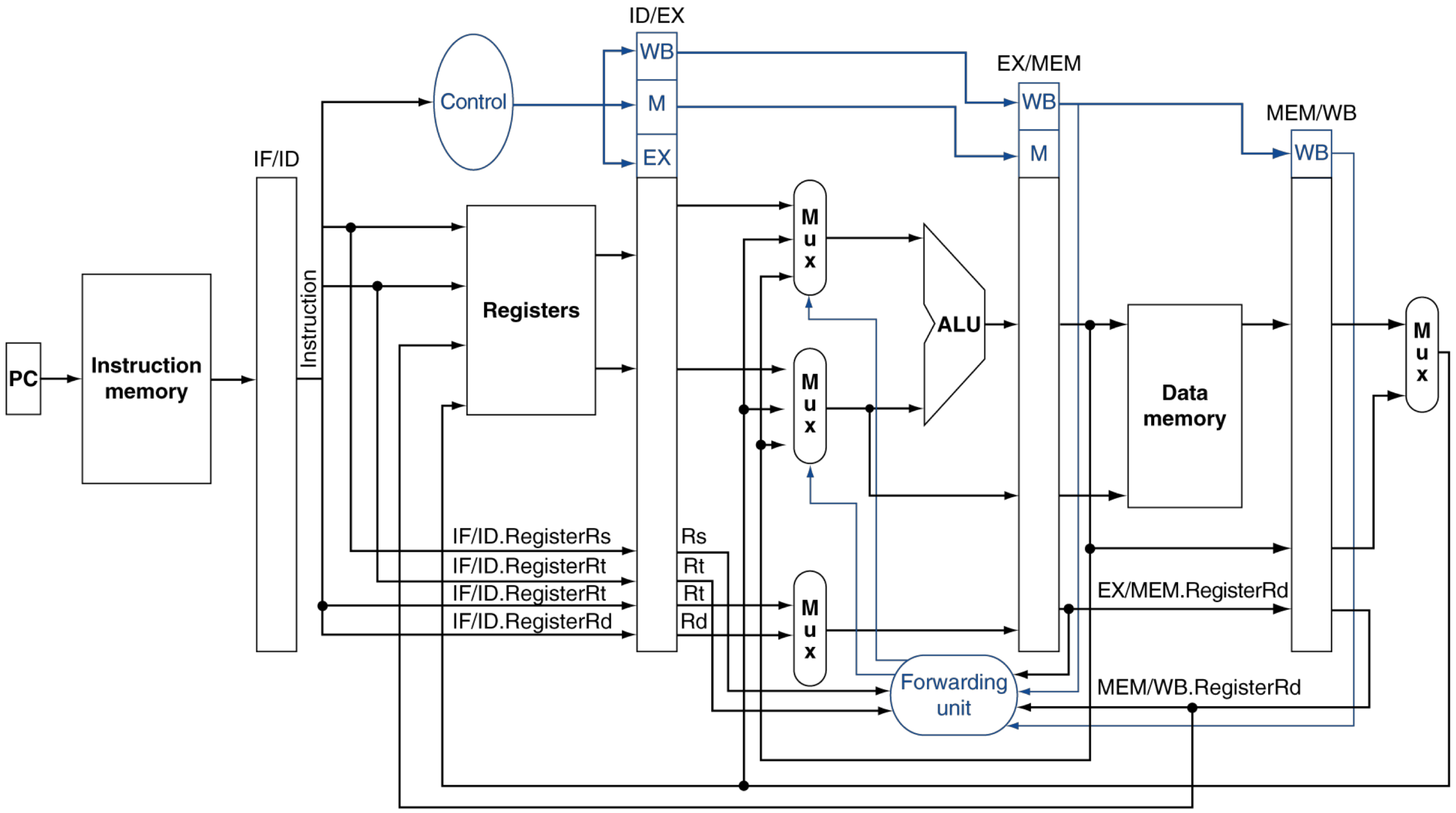
```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

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

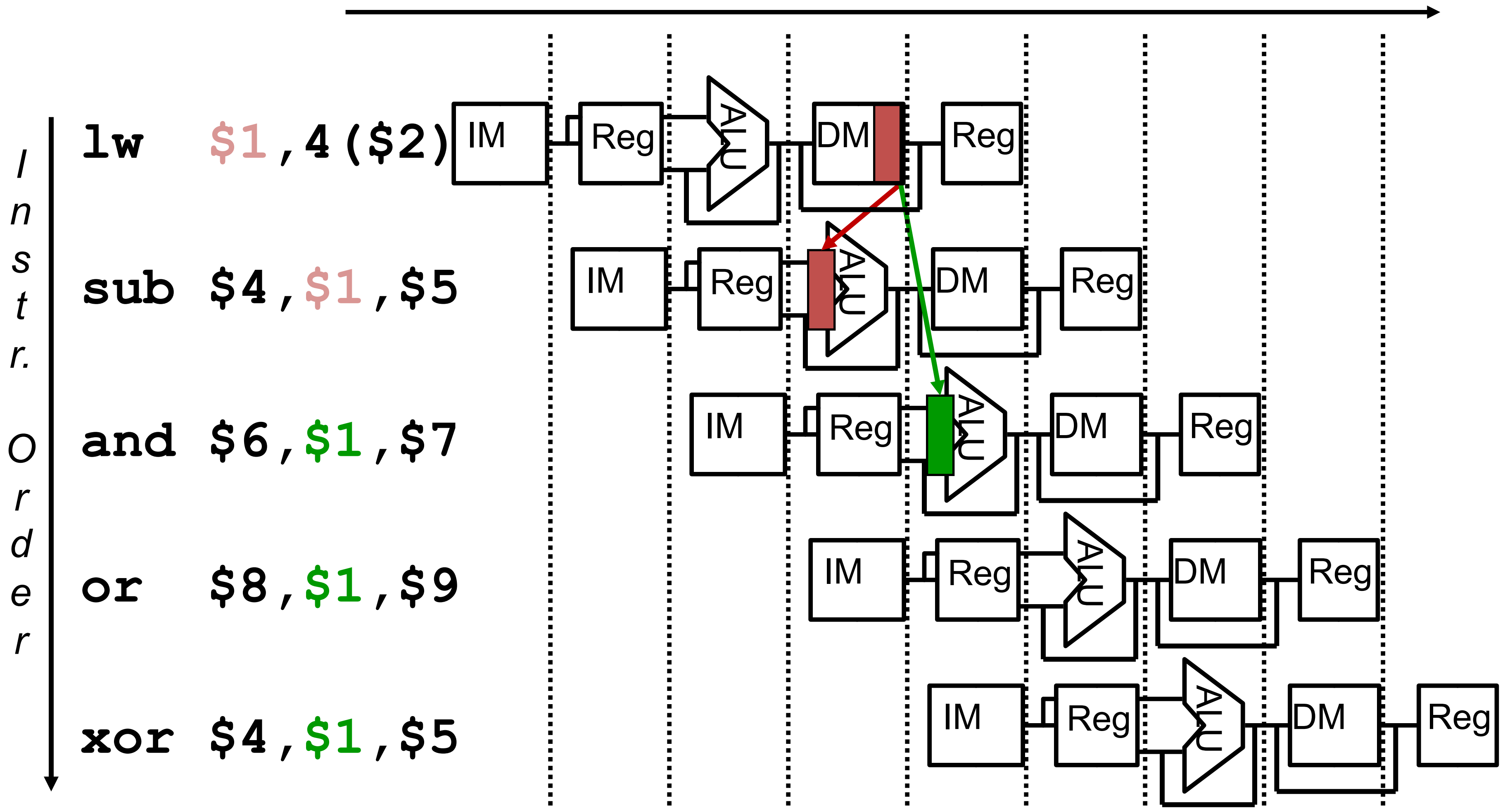
# Forwarding Conditions (cont.)

- MEM hazard
  - 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 = 01
  - 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 = 01

# Datapath with Forwarding



# Review: Forwarding with Load-use Data Hazards

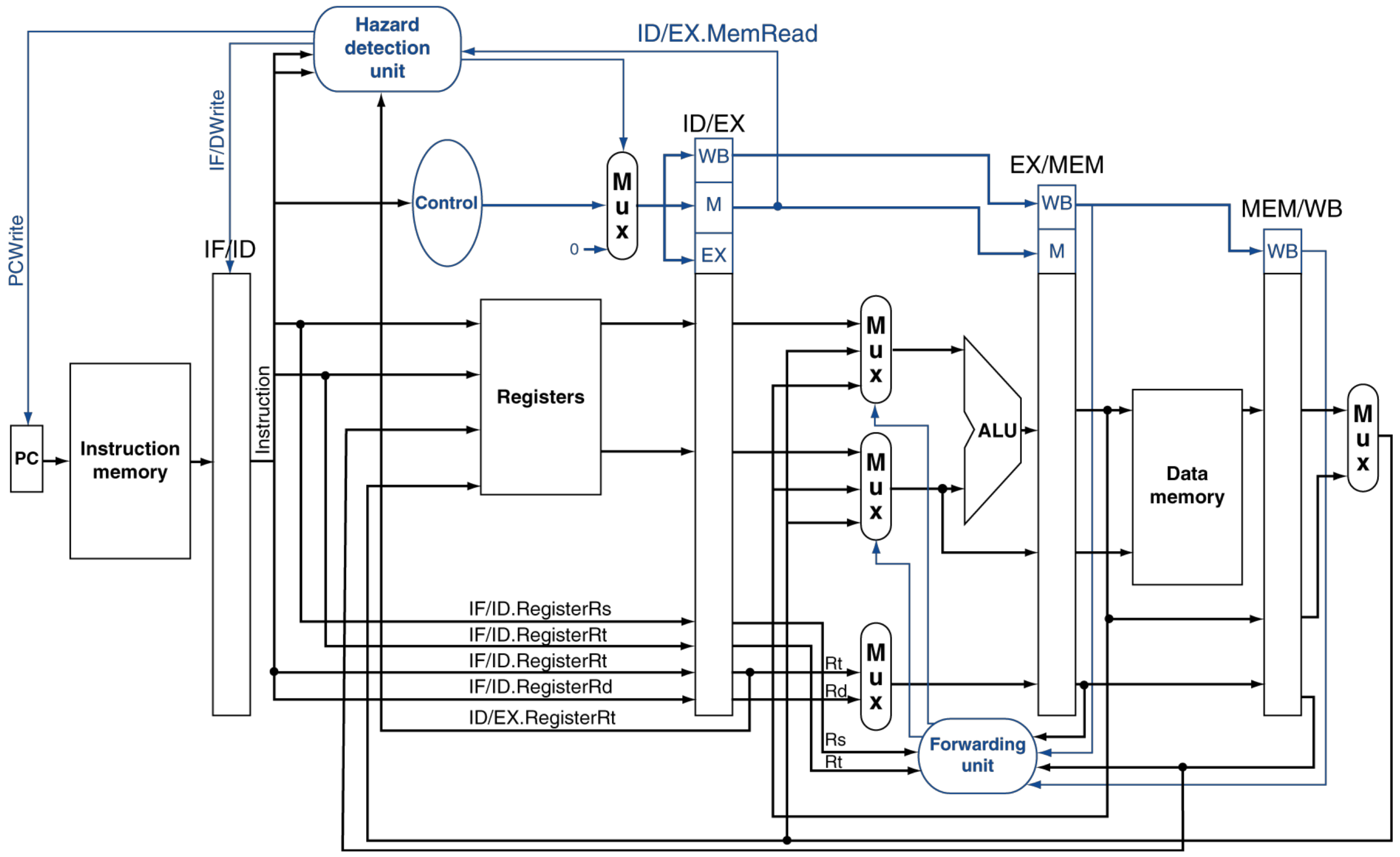


- Will still need one stall cycle even with forwarding

# Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

# Datapath with Hazard Detection



## Forwarding Performance Speedup

```
1: add $1, $2, $3
2: add $3, $2, $1
3: add $2, $2, $1
4: add $1, $3, $3
```

**In-class Assessment!**

**Access Code: PF**

**Note: sharing access code to those outside of classroom or using access while outside of classroom is considered cheating**

## Forwarding Performance Speedup

```
1: add $1, $2, $3
2: add $3, $2, $1
3: add $2, $2, $1
4: add $1, $3, $3
```

# Compilers and Data Hazards

- Compilers rearrange code to try to fill slots with useful stuff
  - Fill load delay slot with a good instruction
  - When successful, the slot has no cost
    - The next instruction does not depend on load result
    - Does not need to stall
    - Show the advantage of the pipeline
  - When can't fill the slot
    - Need to output a NOP if there is no hardware interlock

- Since the pipeline is very machine dependent
  - Need hardware interlocks to run old code
  - Most machines have interlocks!
  - Microprocessor without Interlocked Pipeline Stages

# Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)