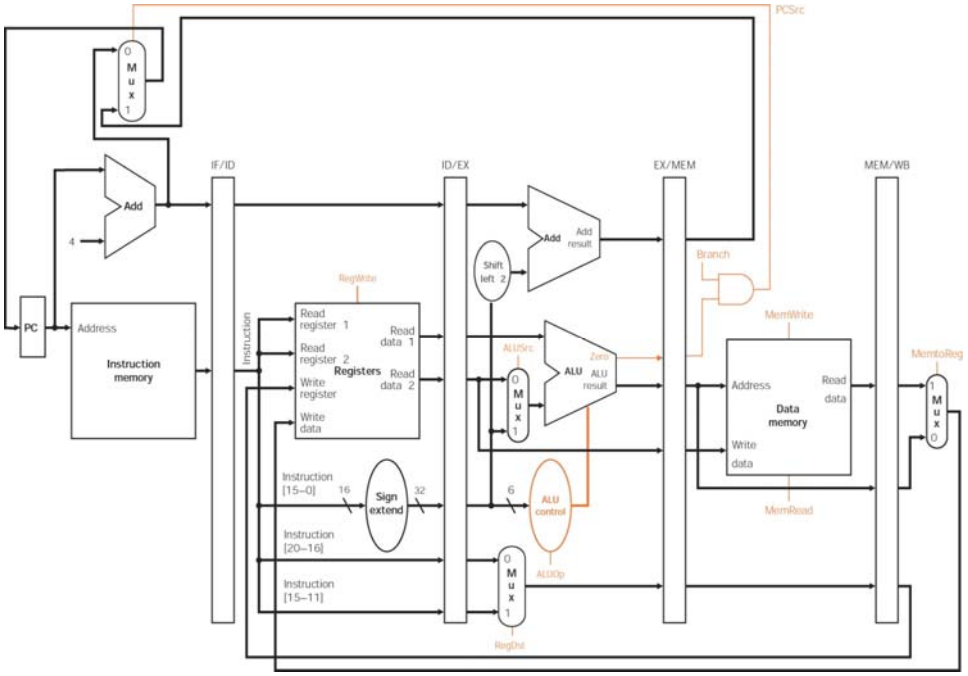
# Chapter Four – II (3/5)



## Pipeline control

- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back
- How would control be handled in an automobile plant?
  - a fancy control center telling everyone what to do?
  - should we use a finite state machine?

# Datapath with Control Signals






# Pipeline Control: ALU Control Signals

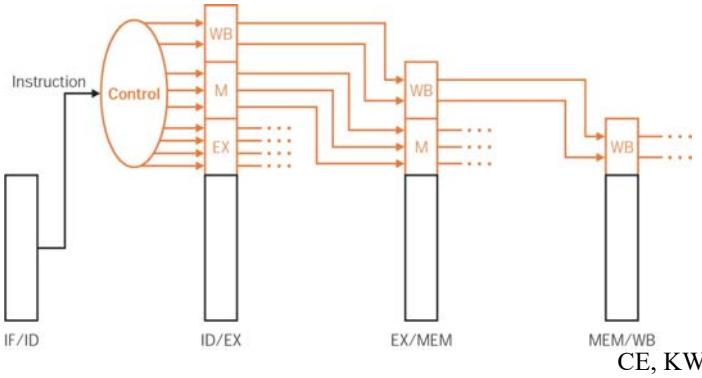
| ALU control input | Function |
|---|---|
| 000 | AND |
| 001 | OR |
| 010 | Add |
| 110 | Sub |
| 111 | Slt |

| Op | ALUOp | Operation | Funct | ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | Load word | XXXXXX | add | 010 |
| SW | 00 | Store word | XXXXXX | add | 010 |
| BEQ | 01 | Branch equal | XXXXXX | sub | 110 |
| R-type | 10 | add | 100000 | add | 010 |
| R-type | 10 | sub | 100010 | sub | 110 |
| R-type | 10 | AND | 100100 | AND | 000 |
| R-type | 10 | OR | 100101 | OR | 001 |
| R-type | 10 | slt | 101010 | slt | 111 |

# Pipeline Control

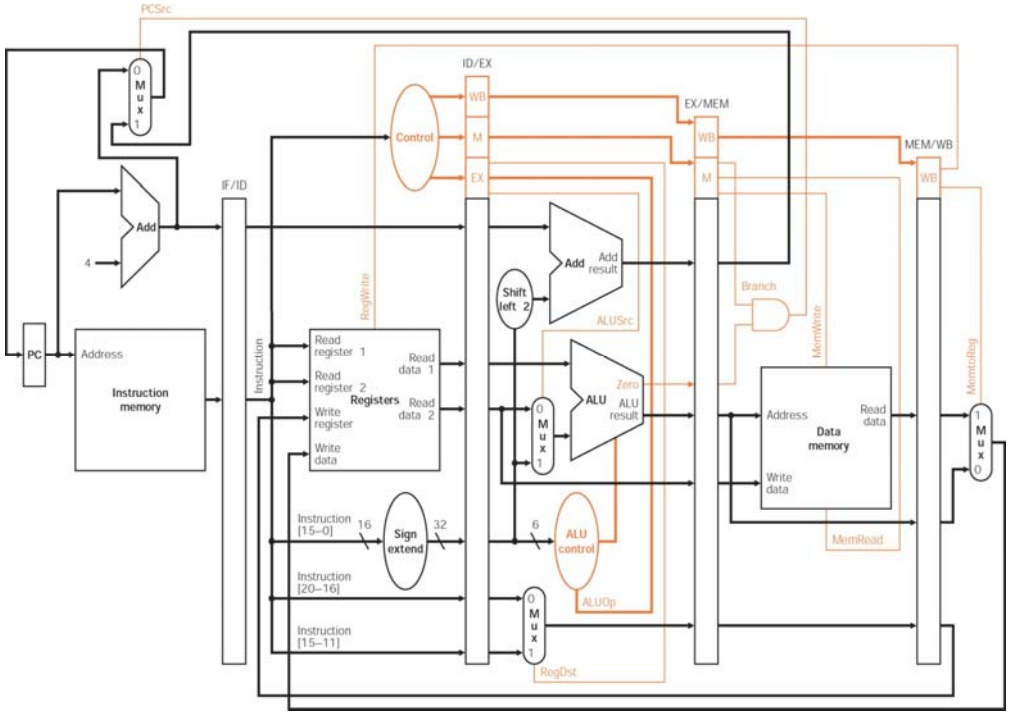
- **Pass control signals along just like the data**

| Instruction | Execution/Address Calculation stage control lines | | | | Memory access stage control lines | | | Write-back stage control lines | |
|---|---|---|---|---|---|---|---|---|---|
| | Reg Dst | ALU Op1 | ALU Op0 | ALU Src | Branch | Mem Read | Mem Write | Reg write | Mem to Reg |
| R-format | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| sw | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| beq | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

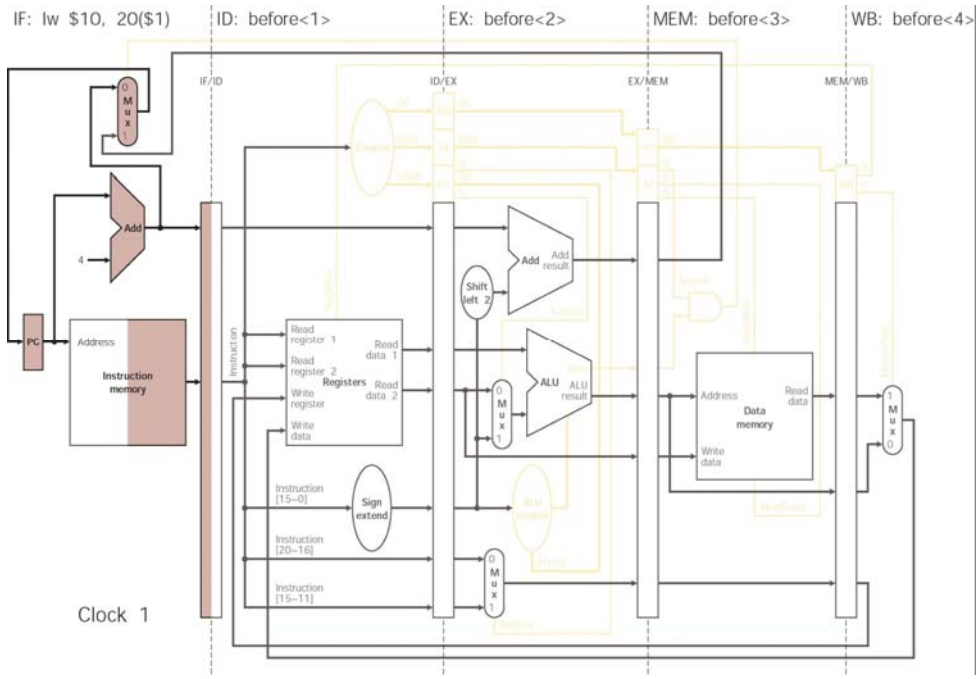





# Datapath with pipelined control signals

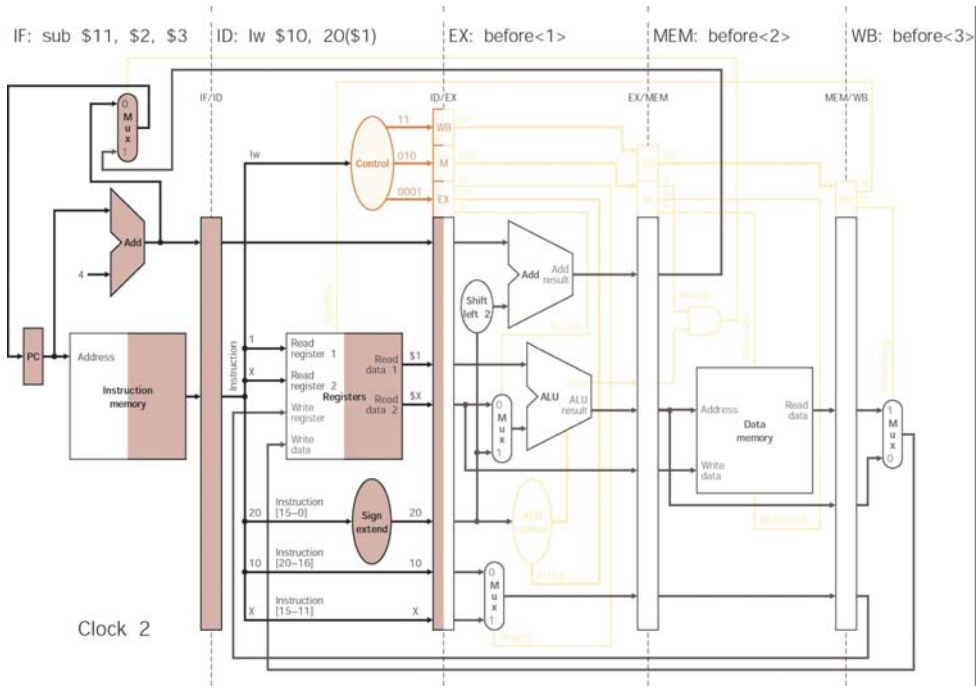


# Execution Example (Clock 1)

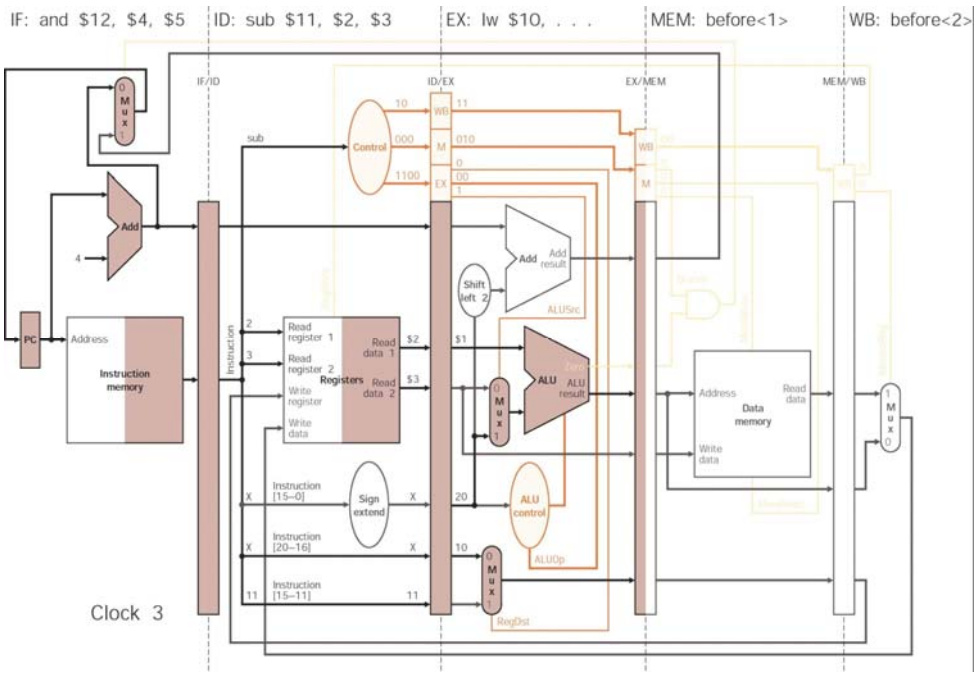





# Execution Example (Clock 2)

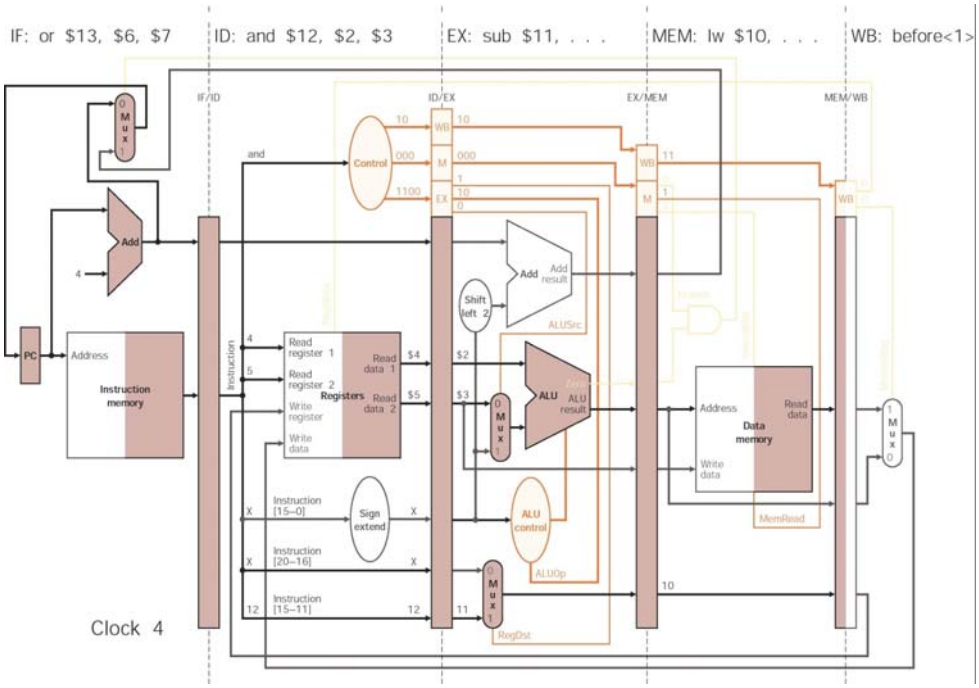


# Execution Example (Clock 3)

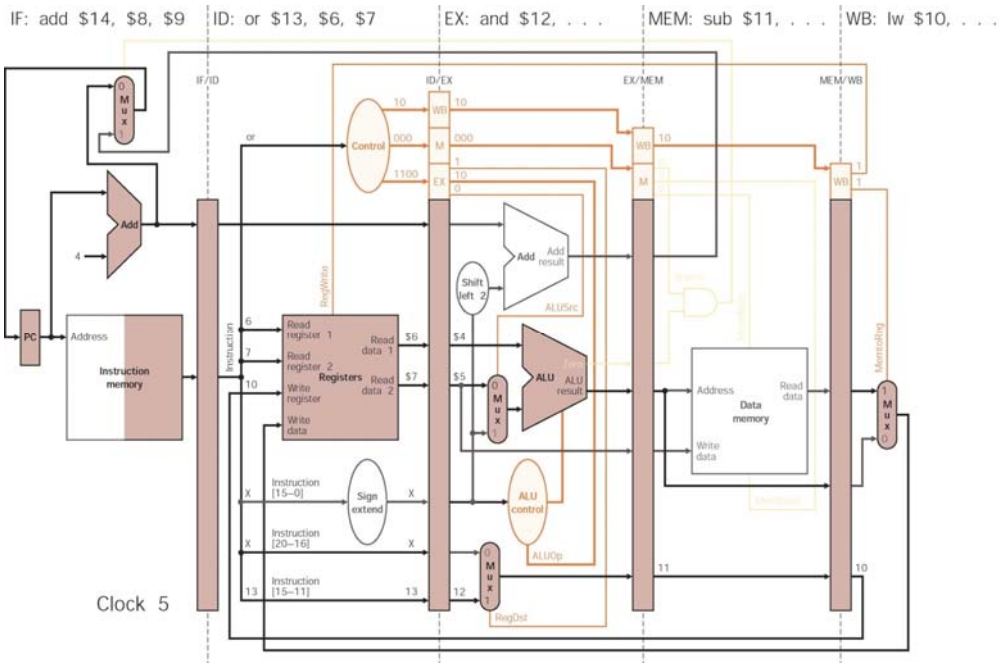





# Execution Example (Clock 4)

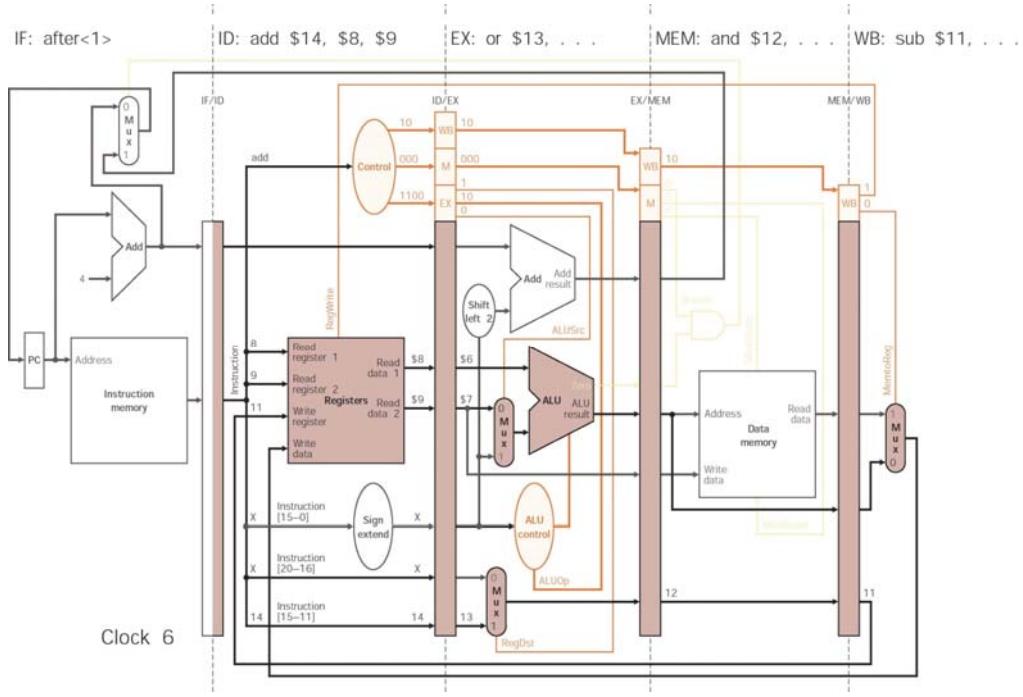


# Execution Example (Clock 5)

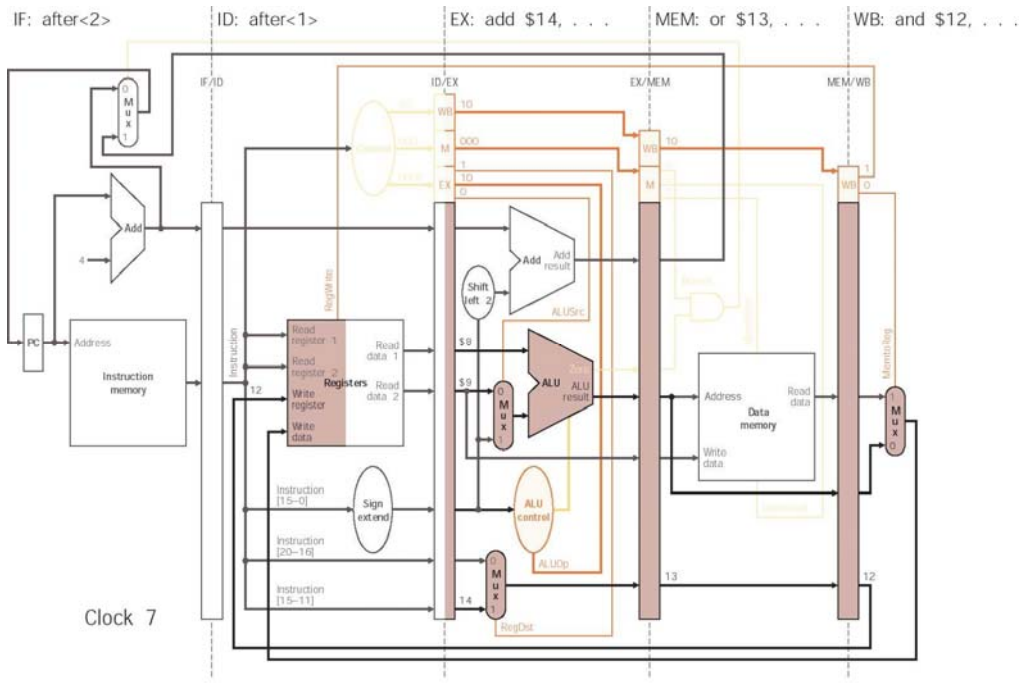





# Execution Example (Clock 6)

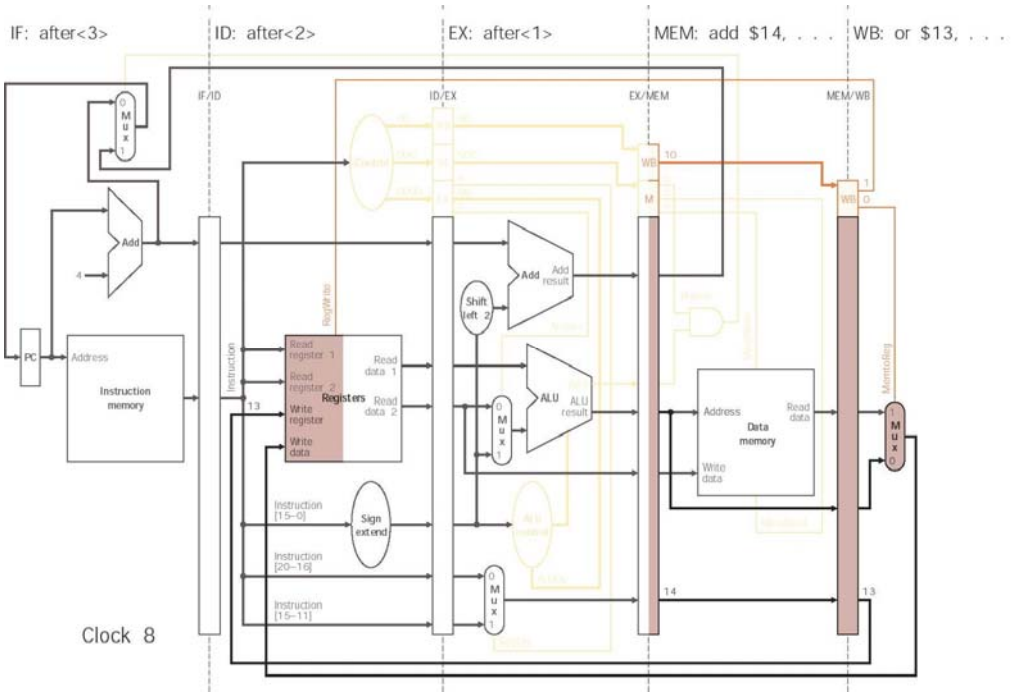


# Execution Example (Clock 7)

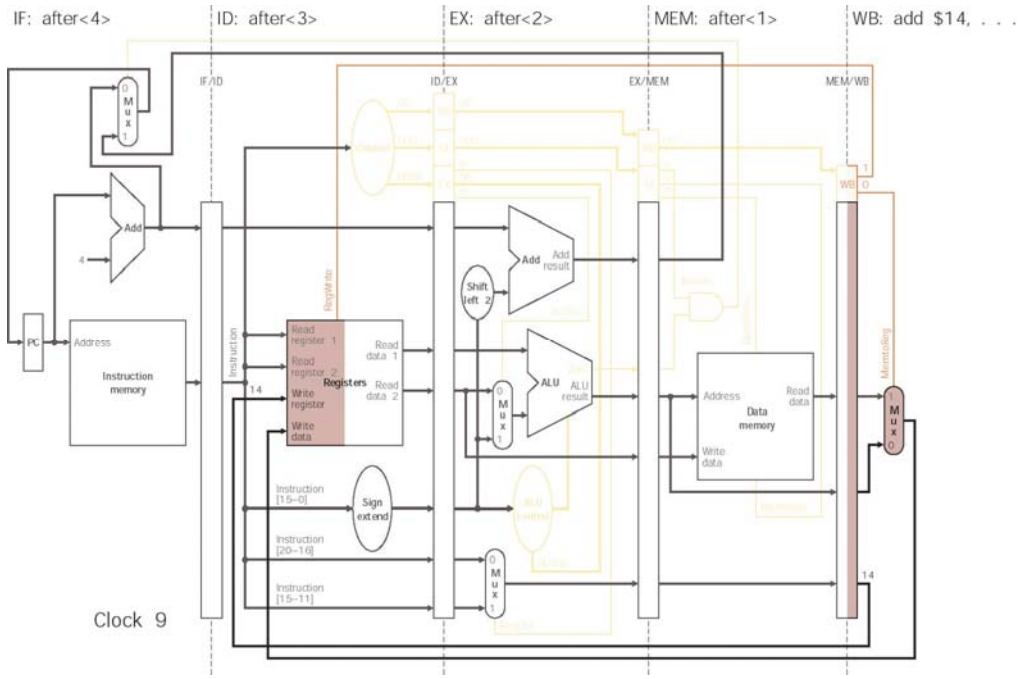





# Execution Example (Clock 8)

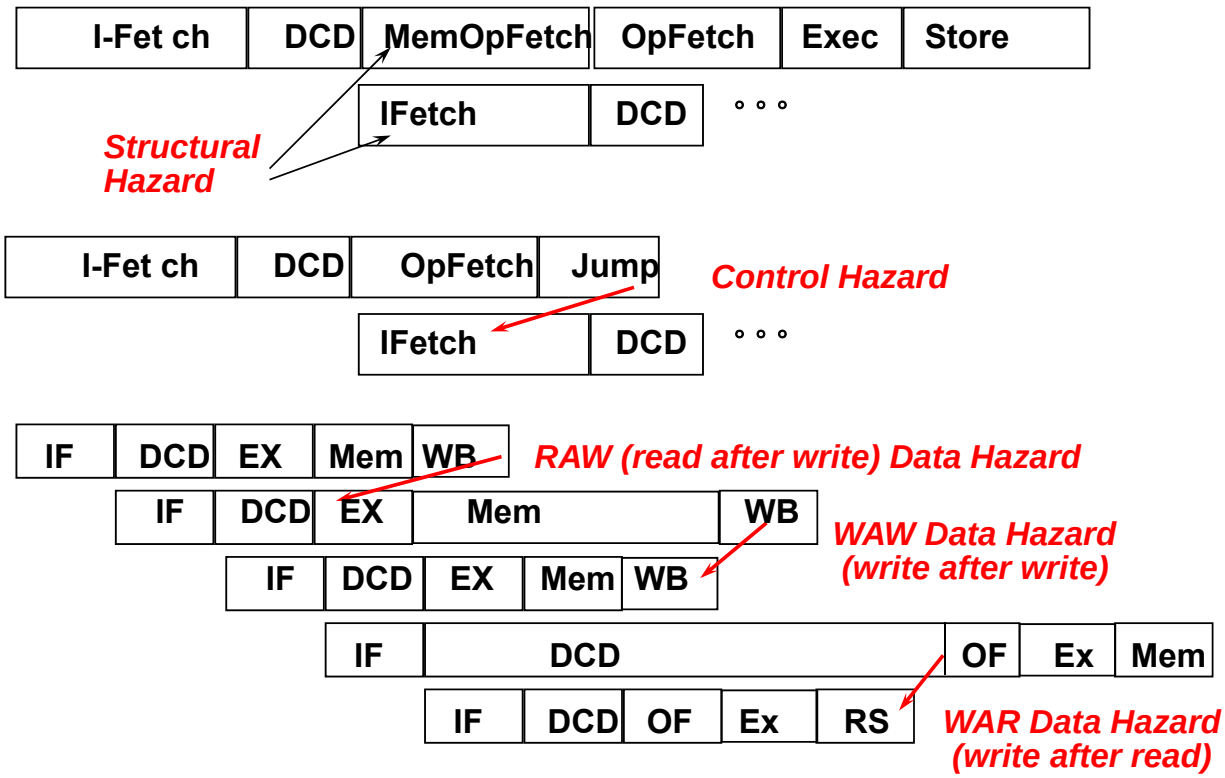


# Execution Example (Clock 9)



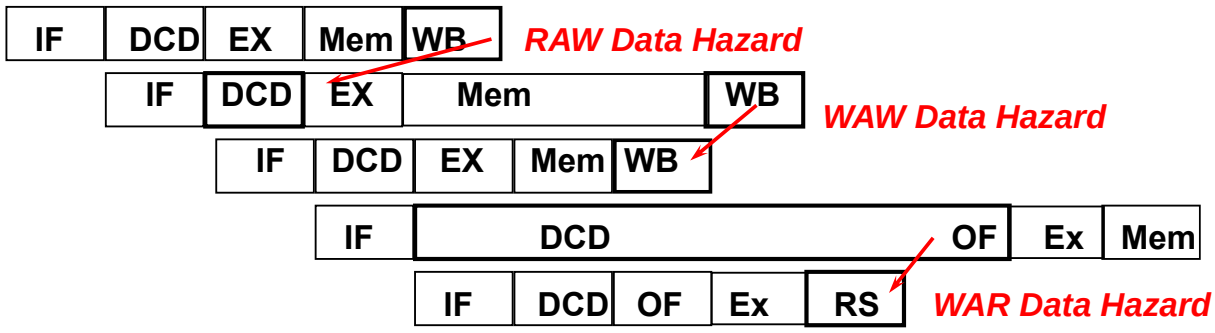



# Pipeline Hazards Again



# Data Hazards

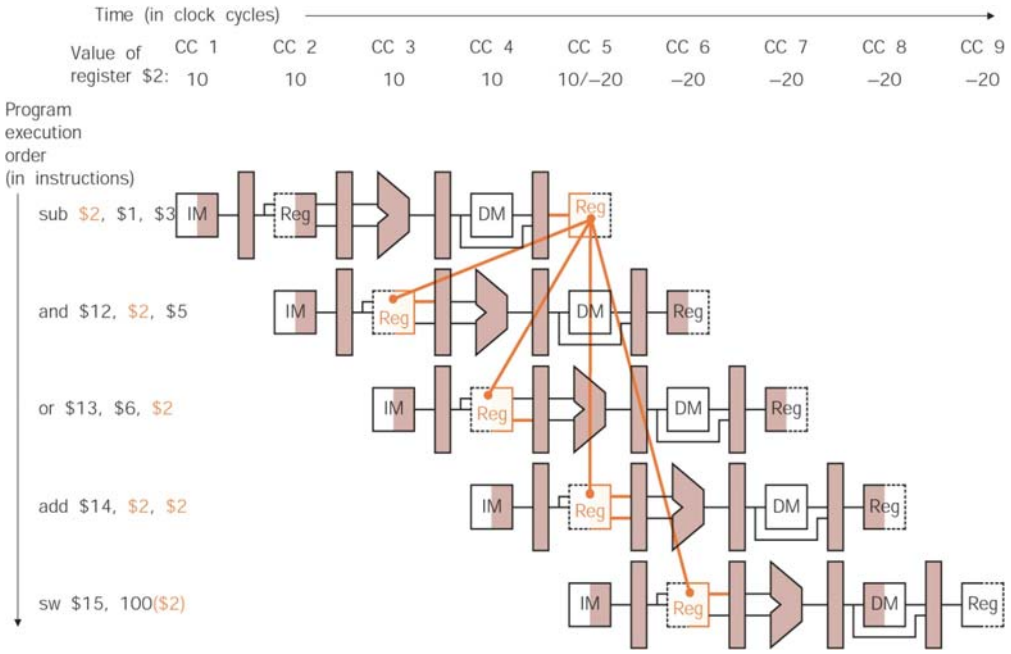
- Avoid some "by design"
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eleminate WAW by doing all WBs in order (last stage, static)
- Detect and resolve remaining ones
  - stall or forward (if possible)






# Dependencies

- Problem with starting next instruction before first is finished
  - dependencies that "go backward in time" are data hazards



# Software Solution

- Have compiler guarantee no hazards
- Where do we insert the “nops” ?

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

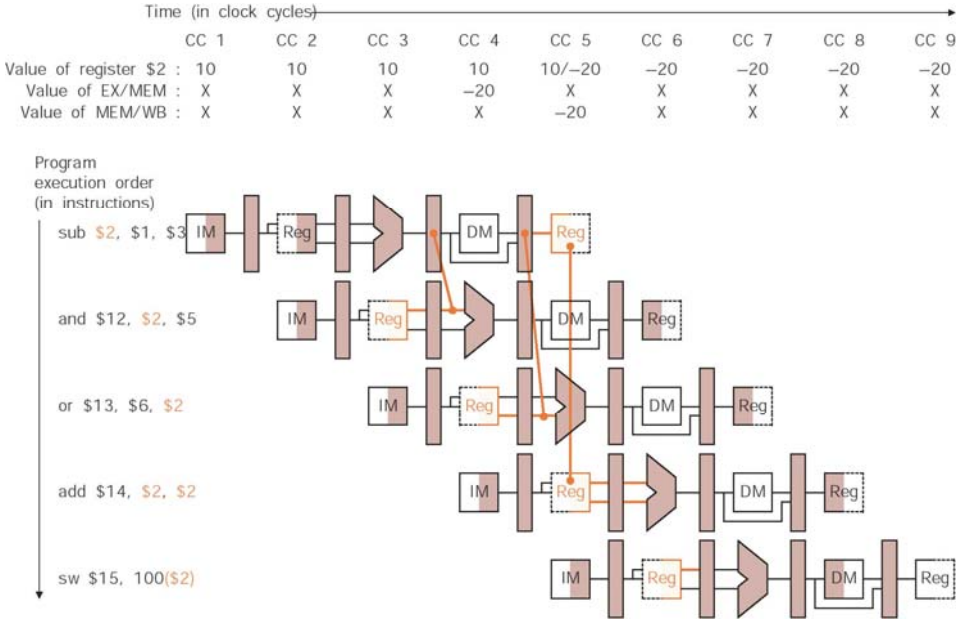
nop
nop
nop

- Problem: this really slows us down!

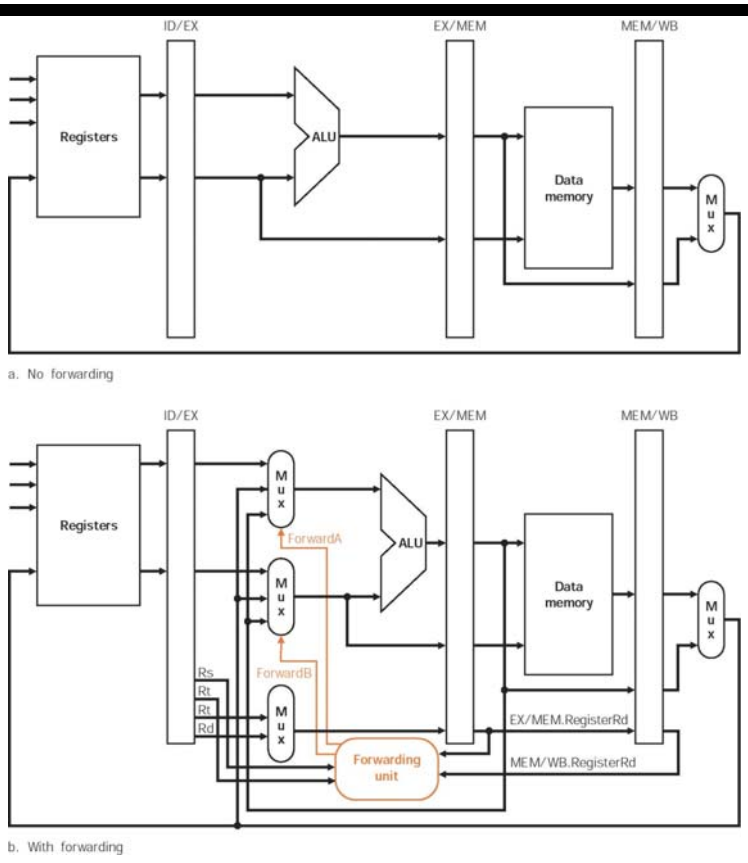


# Forwarding

- Use temporary results, don’t wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding



# Extra hardware for Forwarding






# Hazard Detection

- **Suppose instruction *i* is about to be issued and a predecessor instruction *j* is in the instruction pipeline.**
- **A RAW hazard exists on register *r* if** $r \in Rregs(i) \cap Wregs(j)$
  - **Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.**
  - **When instruction issues, reserve its result register.**
  - **When on operation completes, remove its write reservation.**
- **A WAW hazard exists on register *r* if** $r \in Wregs(i) \cap Wregs(j)$
- **A WAR hazard exists on register *r* if** $r \in Wregs(i) \cap Rregs(j)$

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

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

- 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 (for MIPS)
  - EX/MEM.RegisterRd ≠ 0,
    MEM/WB.RegisterRd ≠ 0



# 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

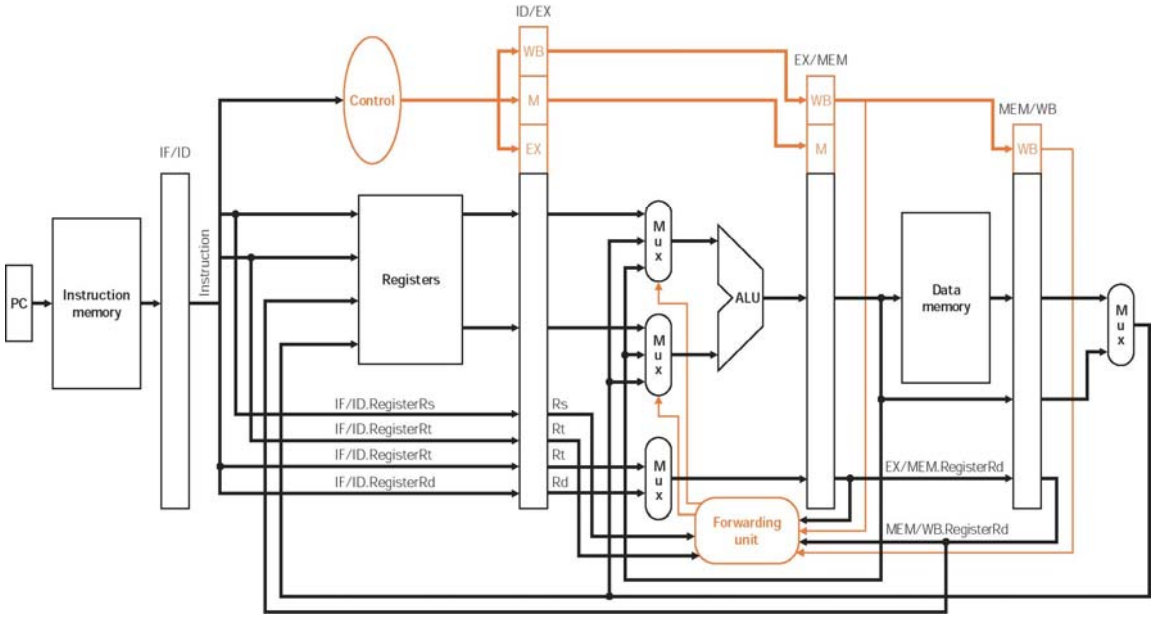
- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true



# Revised Forwarding Condition

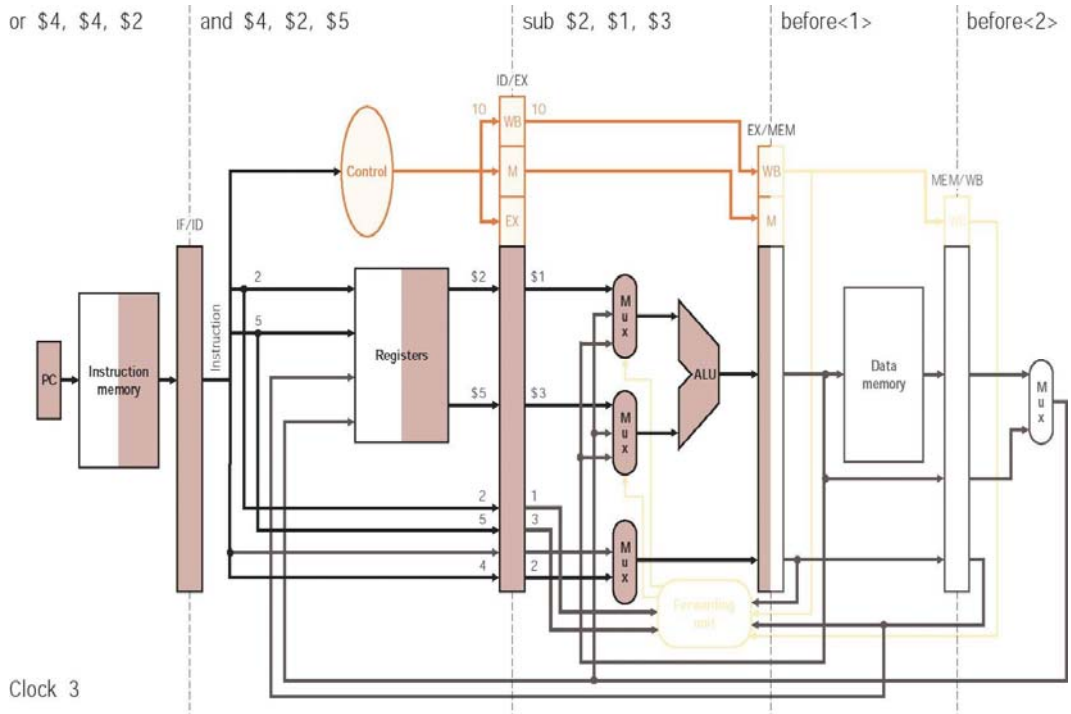
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

# Datapath supporting Forwarding

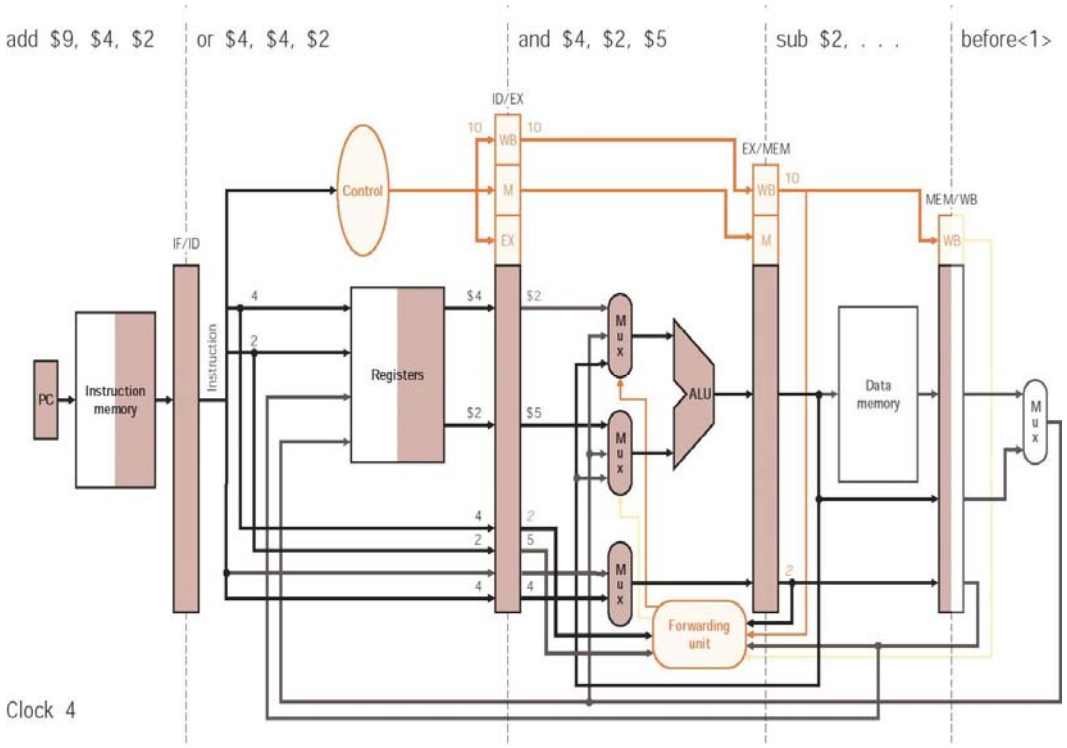





# Modified Execution Example with Forwarding (C3)

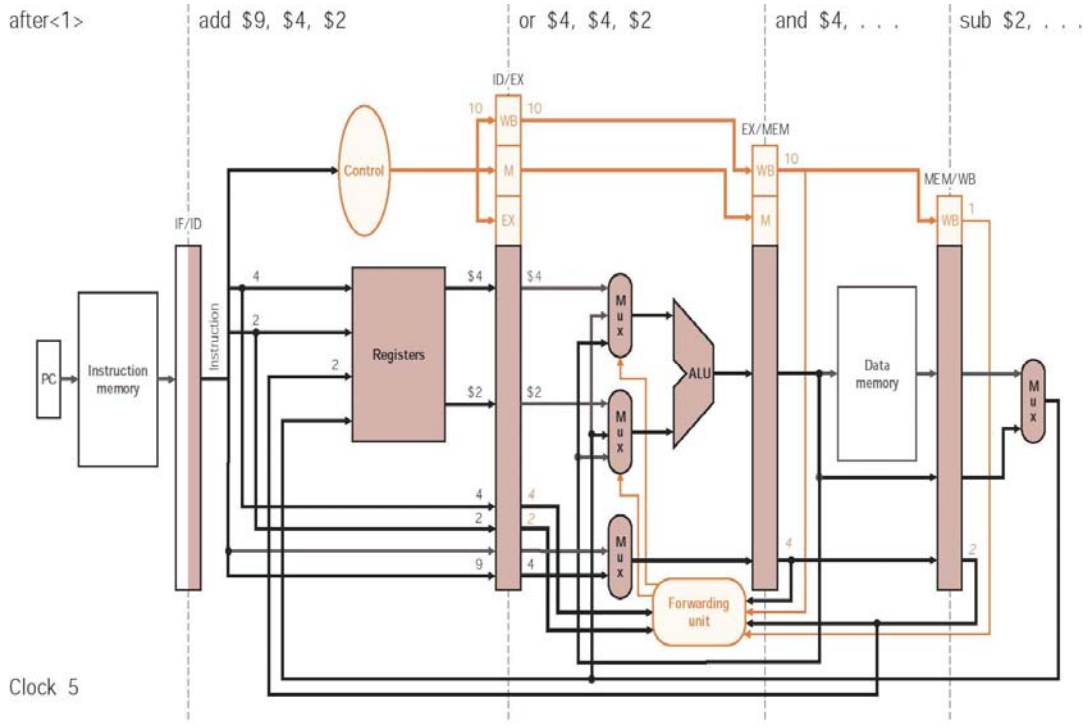


# Modified Execution Example with Forwarding (C4)

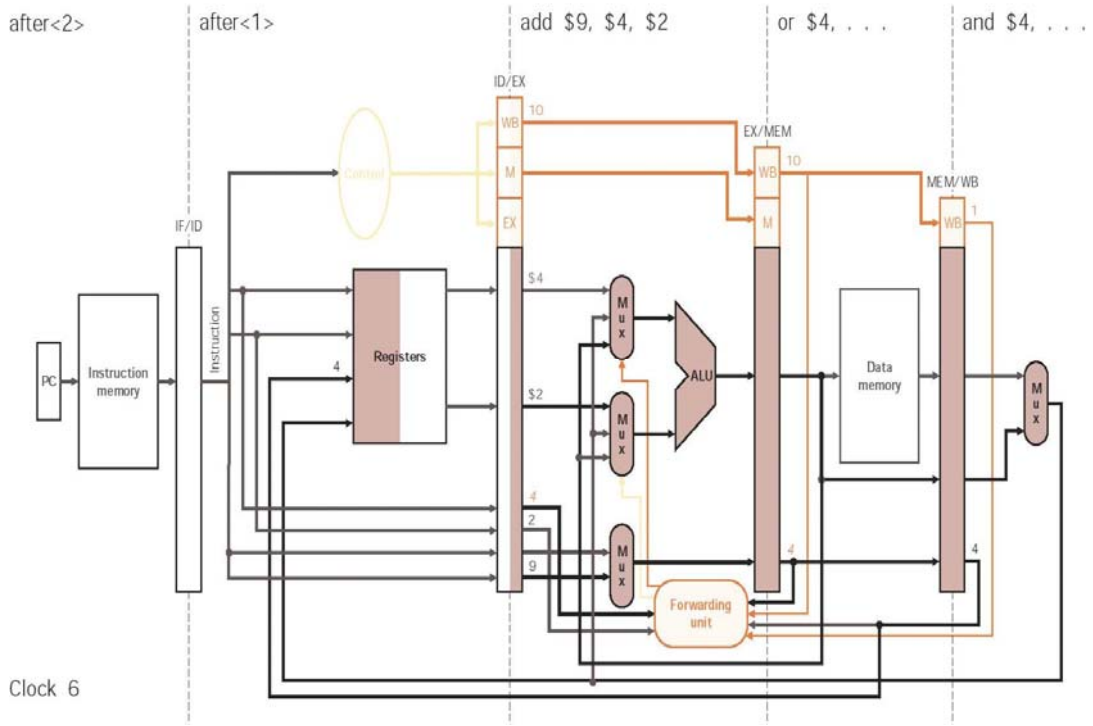





# Modified Execution Example with Forwarding (C5)

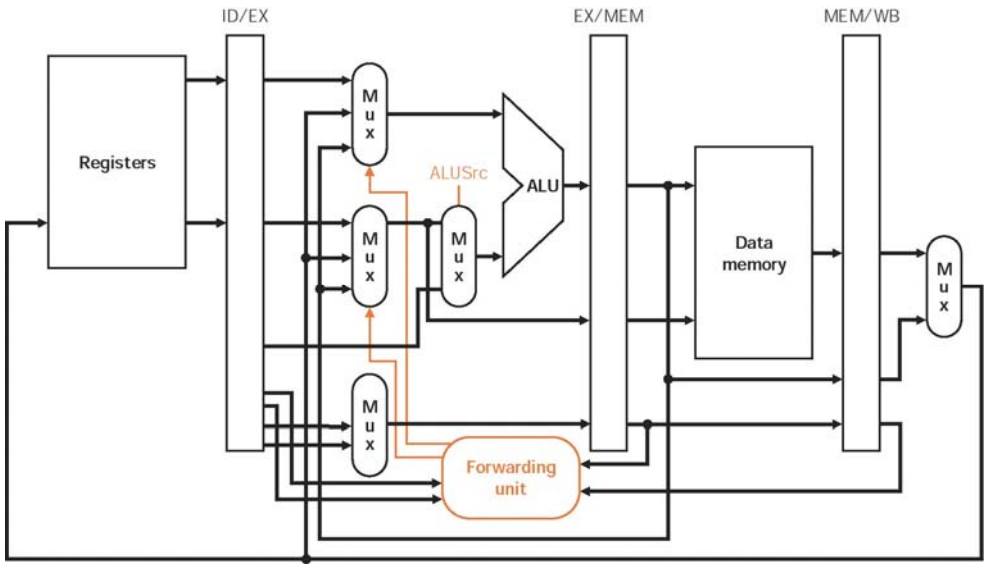


# Modified Execution Example with Forwarding (C6)





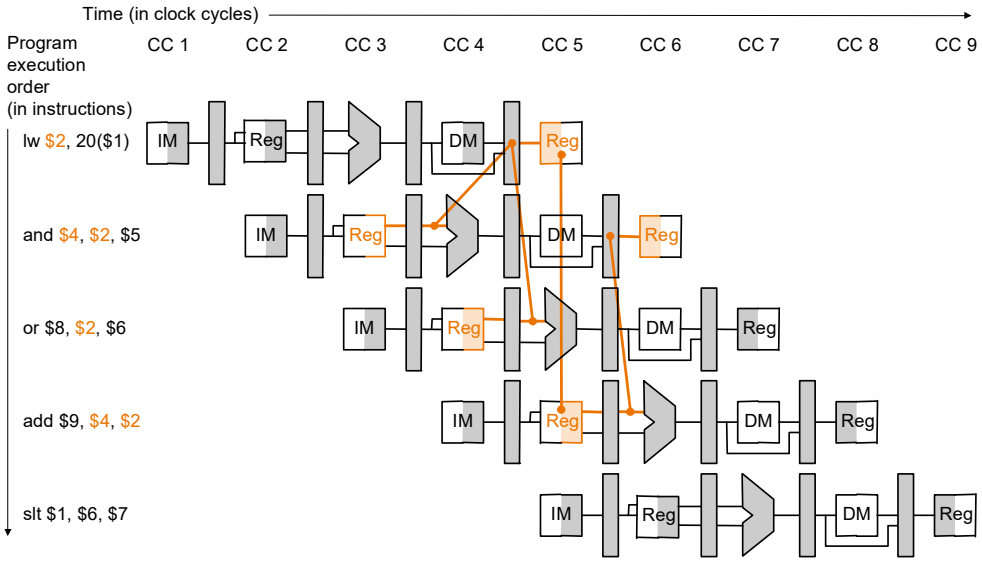

# Added MUX for selecting ALU Src B




- **For signed immediate**

# Can't always forward

- **Load word can still cause a hazard:**
  - **an instruction tries to read a register following a load instruction that writes to the same register.**

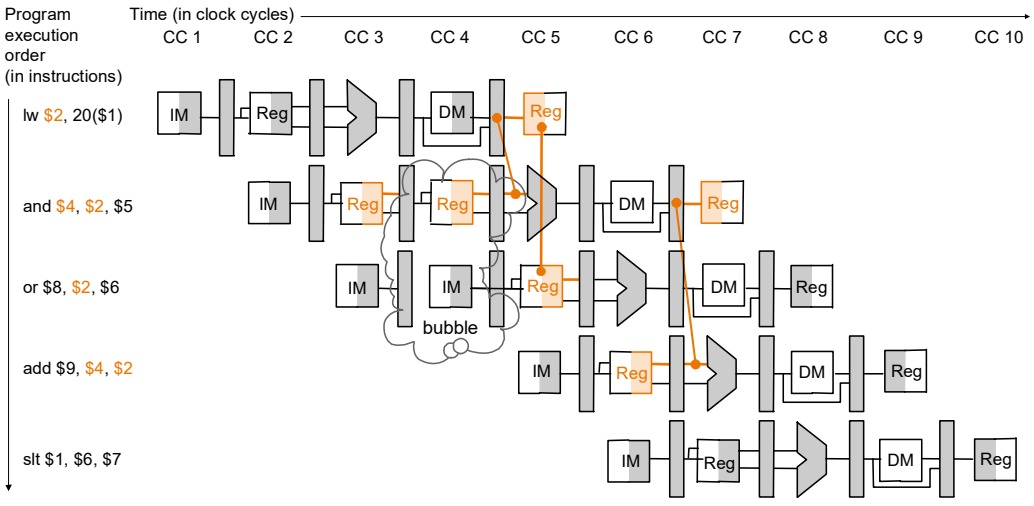



- **Thus, we need a hazard detection unit to "stall" the load instruction**



# Stalling

- **We can stall the pipeline by keeping an instruction in the same stage**



# 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