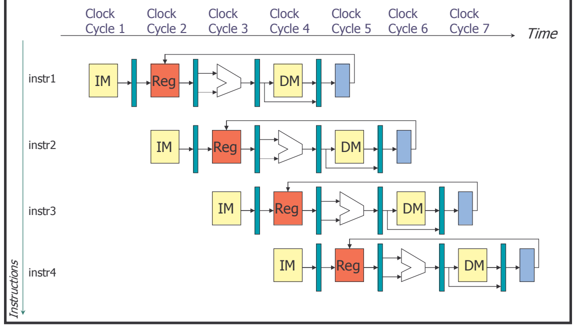
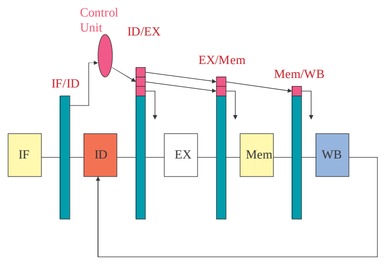
Pipelining review

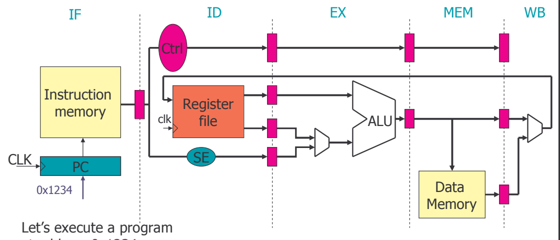
Pipelining review



Pipeline Control – big picture



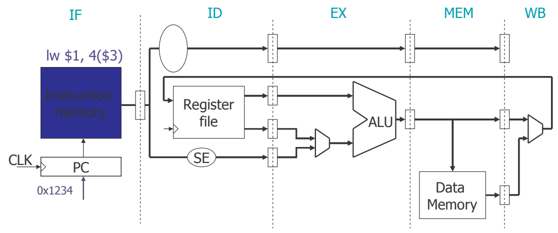
Pipeline in action



Let's execute a program at address 0x1234:
lw $1, 4($3)
add $2, $3, $4
or $3, $1, $2
bne $1, $3, L

Suppose: $1 = 1, $2 = 2, $3 = 1000, $4 = 6

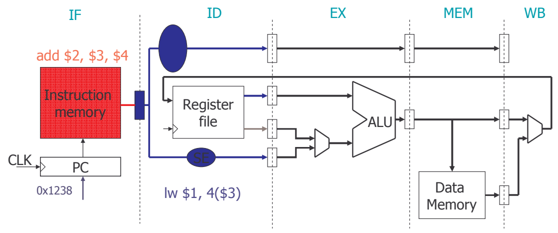
Pipeline in action 1



Clock:
1

$1 = 1, $2 = 2, $3 = 1000, $4 = 4

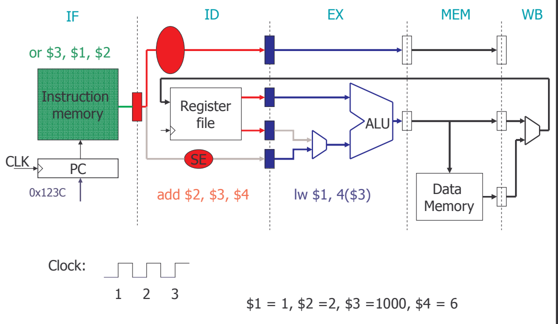
Pipeline in action 2



Clock:
1 2

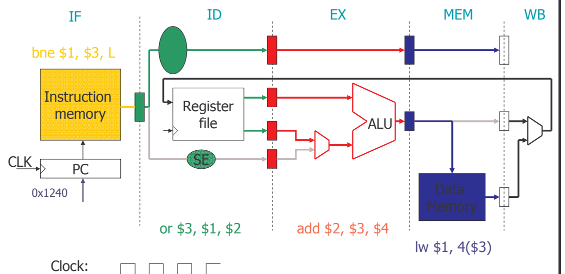
$1 = 1, $2 =2, $3 =1000, $4 = 6

Pipeline in action 3



$1 = 1, $2 =2, $3 =1000, $4 = 6

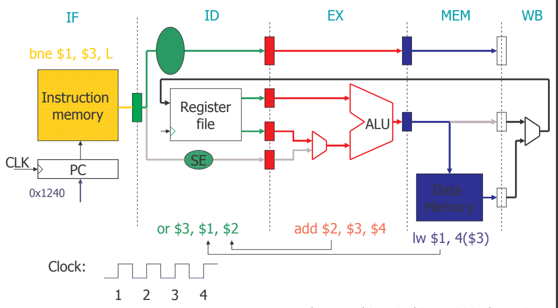
Pipeline in action 4 (anything wrong?)



Let's assume memory returns 42 for lw.

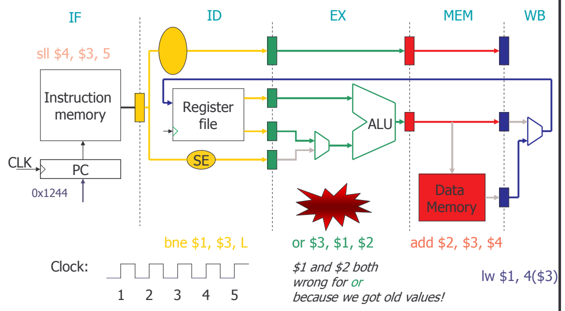
$1 = 1, $2 =2, $3 =1000, $4 = 6

Data Hazards!



$1 = 1, $2 = 2, $3 = 1000, $4 = 6

Pipeline in action 5



$1 and $2 both wrong for or because we got old values!

$1 = 42, $2 = 2, $3 = 1000, $4 = 6

Data Hazards

- Data dependence
  - result of an operation needed before it is stored back in reg. file:

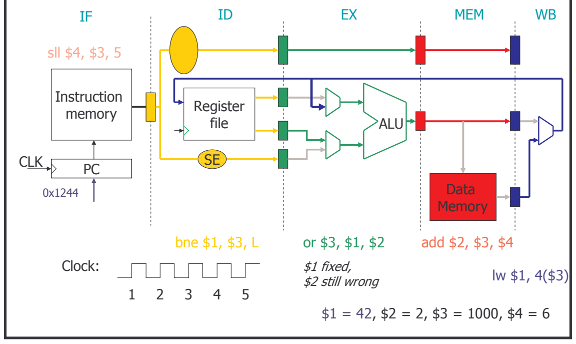
```
lw   $1, 4($3)
add  $2, $3, $4
or   $3, $1, $2
```

  - The data hazard above is read after write (RAW)
- Data dependence (RAW) occurs when:
  - An instruction wants to read a register in stage 2, and
  - One instruction in stage 3 or stage 4 is going to write that register
    - Note: if the instruction writing the register is in stage 5, this is fine since we can write a register and read it in the same cycle
- Hazard detection unit compares register fields across stages.

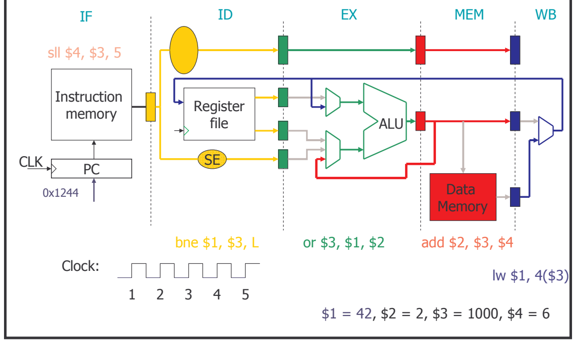
Resolving data dependencies

- Have the compiler generate no-ops
  - Don't have to deal with data hazards in hardware.
- Stall the pipeline, i.e., create *bubbles*
  - the resulting delays are the same as for no-ops
- Send the result generated in stage 3 or stage 4 to the appropriate input of the ALU.
  - This is *forwarding* or *bypassing*.
  - More performance at the cost of more hardware
    - For one simple pipeline, cost is slightly more control and extra buses
    - For several pipelines, say n, communication grows as O(n^2)

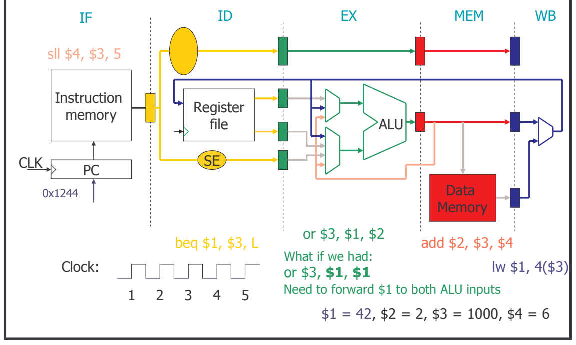
Forwarding from WB to EX



Forwarding from MEM to EX



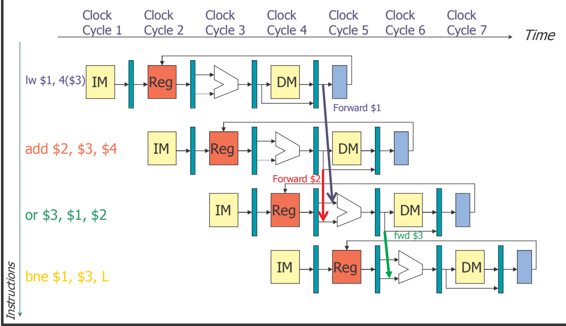
Generally, need to forward to both inputs!



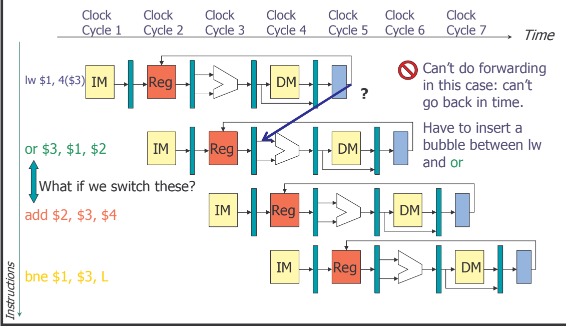
Forwarding a register twice?

- What if had:
  - add $1, $2, $2
  - or $1, $3, $4
  - and $5, $1, $1
- Where do you forward $1 from?

Another look at forwarding…



Data hazards and loads

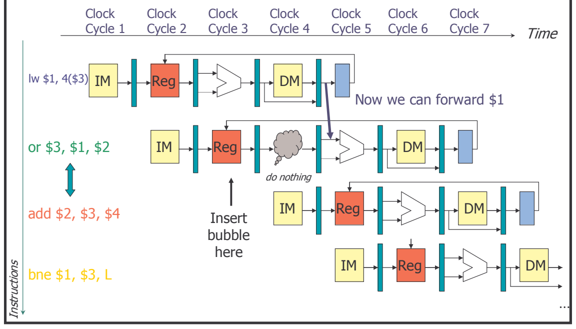


Can't do forwarding in this case: can't go back in time.

Have to insert a bubble between lw and or

Inserting a bubble

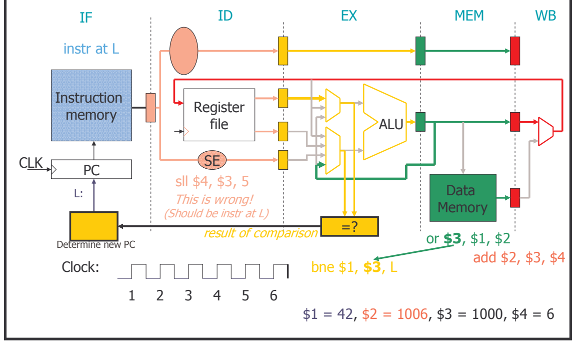
- Have to do it when you load into a register which is used in the next instruction



Other hazards

- We've seen data hazards
  - when an instruction in the pipeline is *dependent* on another instruction still in the pipeline.
  - Resolved by:
    - *Bypassing/forwarding*
    - *Stalling*
- Other types of hazards:
  - *Structural hazards*
    - where two instructions at different stages want to use the same resource.
    - Solved by using more resources (e.g., instruction and data memory; several ALU's). Won't happen in our pipeline.
  - *Control hazards*
    - happens on a taken branch.
    - Evaluation of branch condition and calculation of branch target is not completed before next instruction is fetched.

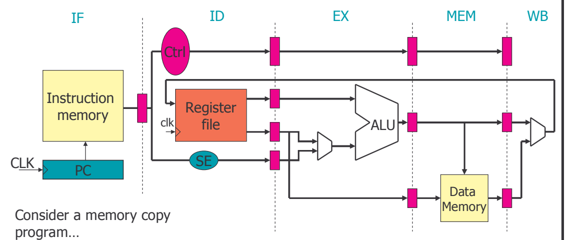
Branch path



Resolving control hazards

- *Stall* until result of the condition & target are known
  - too slow
- Reduce penalty by redesigning the pipeline:
  - move branch calculation to ID stage
  - move branch comparison to ID stage
    - how to do quick compare?
  - use both edges of the clock
- Delay slots
  - specify in ISA that instruction after branch is always executed.
- Branch prediction
  - in IF stage, guess branch outcome
  - correct it if turns out to be wrong.

Lw & sw



Consider a memory copy program...

```
loop:   lw $1, 0($3)  }  ←—— Too many stalls!
        sw $1, 0($4)  }
        addi $3, $3, 4       How do we fix this?
        addi $4, $4, 4
        bne $4, $5, Loop
```