Stalling delays the entire pipeline

- If we delay the second instruction, we’ll have to delay the third one too
  - This is necessary to make forwarding work between AND and OR
  - It also prevents problems such as two instructions trying to write to the same register in the same cycle

```
lw   $2, 20($3)
and  $12, $2, $5
or   $13, $12, $2
```
But what about the ALU during cycle 4, the data memory in cycle 5, and the register file write in cycle 6?

Those units aren’t used in those cycles because of the stall, so we can set the EX, MEM and WB control signals to all 0s.
Stall = Nop conversion

- The effect of a load stall is to insert an empty or **nop** instruction into the pipeline.
Detecting Stalls, cont.

- **When should stalls be detected?**
  
  **EX stage**

  ```
  lw $2, 20($3) and $12, $2, $5
  ```

- **What is the stall condition?**

  ```
  if (ID/EX.MemRead = 1 and (ID/EX.rt = IF/ID.rs or ID/EX.rt = IF/ID.rt))
  then stall
  ```
Adding hazard detection to the CPU
Branches in the original pipelined datapath

When are they resolved?

Instruction memory

Read Instruction address [31-0]

RegWrite

Registers

Read register 1 Read data 1
Read register 2 Read data 2
Write register Write data

Shift left 2

ALU

Zero

Result

ALUSrc

ALUOp

MemWrite

Data memory

Address

Write data
Read data

MemRead

MemToReg

0

1
Branches

- Most of the work for a branch computation is done in the EX stage:
  - branch target address is computed
  - source registers are compared by the ALU, and the Zero flag is set or cleared accordingly
- Thus, the branch decision cannot be made until the end of the EX stage
  - But we need to know which instruction to fetch next, in order to keep the pipeline running!
  - This leads to what’s called a control hazard

```
beq $2, $3, Label
```
Stalling is one solution

- Again, stalling is always one possible solution:

```
beq $2, $3, Label
```

- Here we just stall until cycle 4, after we do make the branch decision
Branch prediction

- Another approach is to guess whether or not the branch is taken:
  - In terms of hardware, it’s easier to assume the branch is not taken
  - This way we just increment the PC and continue execution, as for normal instructions
- If we’re correct, then there is no problem and the pipeline keeps going at full speed

```
beq $2, $3, Label
```

Clock cycle

1 2 3 4 5 6 7

next instruction 1

next instruction 2
Branch misprediction

- If our guess is wrong, then we would have already started executing two instructions incorrectly. We’ll have to discard, or flush, those instructions and begin executing the right ones from the branch target address, Label.

```
beq $2, $3, Label

next instruction 1
```

```
next instruction 2
```

```
Label: ...
```
Performance gains and losses

- Overall, branch prediction is worth it:
  - Mispredicting a branch means that two clock cycles are wasted
  - But if our predictions are even just occasionally correct, then this is preferable to stalling and wasting two cycles for every branch

- All modern CPUs use branch prediction
  - Accurate predictions are important for optimal performance
  - Most CPUs predict branches dynamically—statistics are kept at runtime to determine the likelihood of a branch being taken

- The pipeline structure also has a big impact on branch prediction:
  - A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance
  - We must also be careful that instructions do not modify registers or memory before they get flushed
Implementing branches

- We can actually decide the branch a little earlier, in ID instead of EX
  - Our sample instruction set has only a BEQ
  - We can add a small comparison circuit to the ID stage, after the source registers are read
- Then we would only need to flush one instruction on a misprediction

```
beq $2, $3, Label
```

next instruction 1

Label: . . .
Implementing flushes

- We must flush one instruction (in its IF stage) if the previous instruction is BEQ and its two source registers are equal.

- We can flush an instruction from the IF stage by replacing it in the IF/ID pipeline register with a harmless nop instruction.
  - MIPS uses sll $0, $0, 0 as the nop instruction.
  - This happens to have a binary encoding of all 0s: 0000 .... 0000.

- Flushing introduces a bubble into the pipeline, which represents the one-cycle delay in taking the branch.

- The IF.Flush control signal shown on the next page implements this idea, but no details are shown in the diagram.
Branching *without* forwarding and load stalls

The other stuff just won’t fit!
Summary

- Three kinds of hazards conspire to make pipelining difficult
  - **Structural hazards** result from not having enough hardware available to execute multiple instructions simultaneously
    - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages
  - **Data hazards** can occur when instructions need to access registers that haven’t been updated yet
    - Hazards from R-type instructions can be avoided with forwarding
    - Loads can result in a “true” hazard, which must stall the pipeline
  - **Control hazards** arise when the CPU cannot determine which instruction to fetch next
    - We can minimize delays by doing branch tests earlier in the pipeline
    - We can also take a chance and predict the branch direction, to make the most of a bad situation