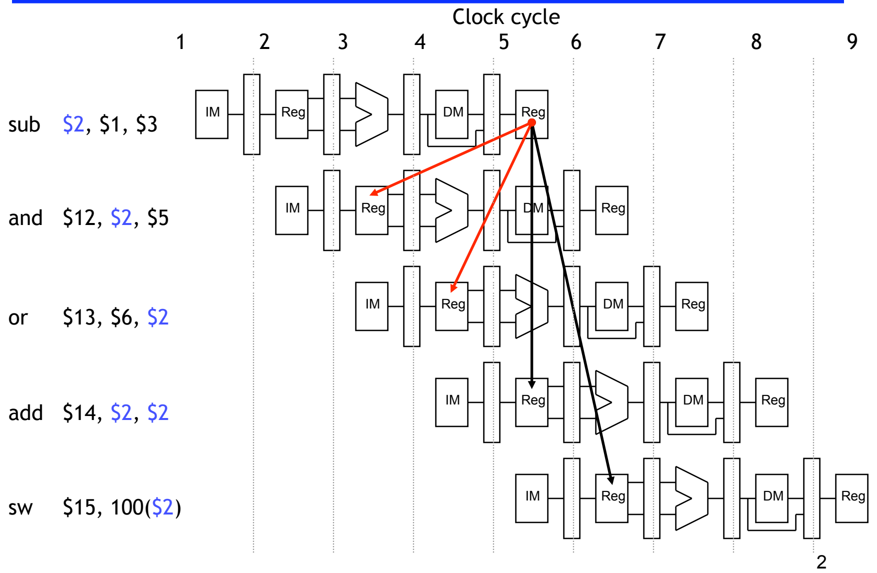
Announcements

- Reading … 5.1, 5.2
- HW 2 due today
- No Class Wednesday

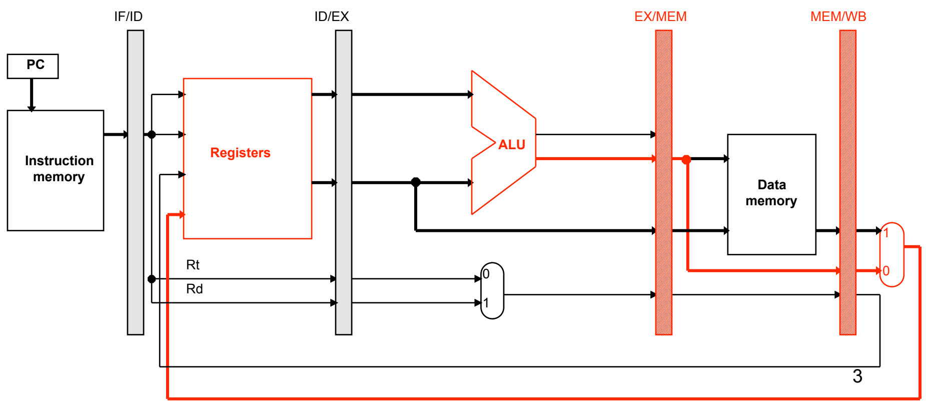
Recall the pipeline diagram



Where to find the ALU result

The ALU result generated in the EX stage is normally passed through the pipeline registers to the MEM and WB stages, before being written to register file

This is an abridged diagram of our pipelined datapath.



3

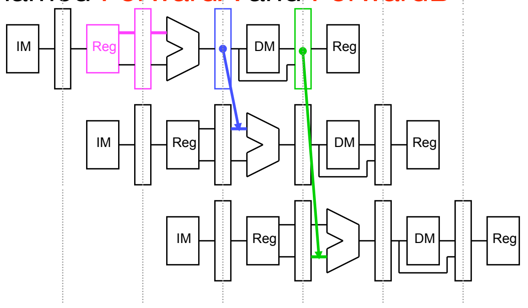
Outline of forwarding hardware

A forwarding unit selects the correct ALU inputs for the EX stage

- ❖ If there is no hazard, the ALU's operands will come from the register file, just like before.
- ❖ If there is a hazard, the operands will come from either the EX/MEM or MEM/WB pipeline registers instead.

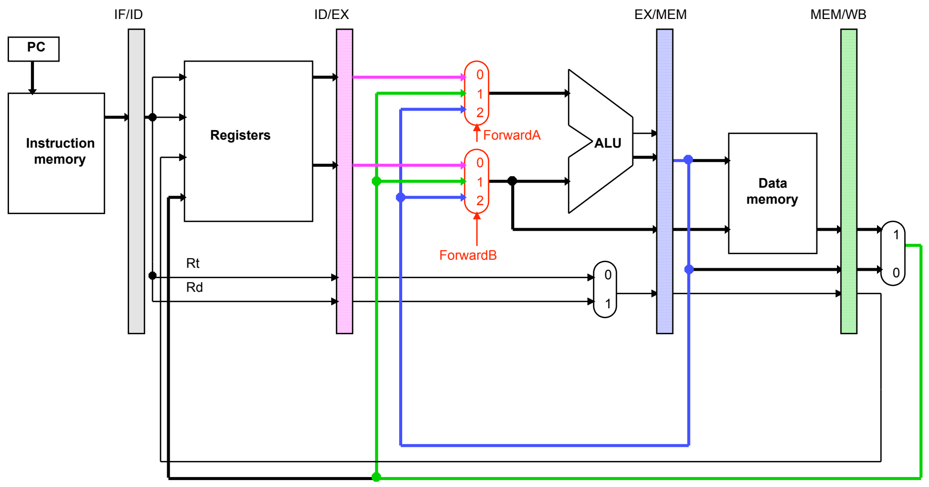
The ALU sources will be selected by 2 new muxes, with control signals named ForwardA and ForwardB

```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
```



4

Simplified datapath w/ forwarding muxes



5

Detecting EX/MEM data hazards

So how can the hardware determine if a hazard exists?

An EX/MEM hazard occurs between the instruction currently in its EX stage and the previous instruction if

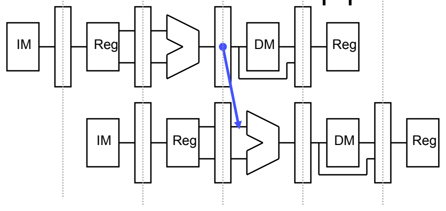
1. The previous instruction will write to the register file, *and*
2. The destination is an the ALU source register in the EX stage

An EX/MEM hazard exists between 2 instructions below

Data in a pipeline register can be referenced using a class-like syntax. For example, ID/EX.RegisterRt refers to the rt field stored in the ID/EX pipeline

```
sub  $2, $1, $3

and  $12, $2, $5
```



6

EX/MEM data hazard equations

The first ALU source comes from the pipeline register when necessary

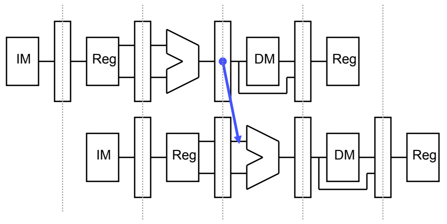
```
if (EX/MEM.RegWrite == 1 && EX/MEM.RegisterRd  == ID/EX.RegisterRs)
        then ForwardA = 2
```

The second ALU source is similar.

```
if (EX/MEM.RegWrite == 1 && EX/MEM.RegisterRd = ID/EX.RegisterRt)
        then ForwardB = 2
```

sub $2, $1, $3

and $12, $2, $5



7

Detecting MEM/WB data hazards

A MEM/WB hazard may occur between an instruction in the EX stage and one from *two* cycles ago

One new problem is if a register is updated twice in a row

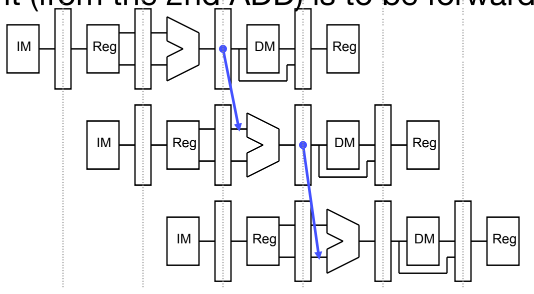
```
add  $1, $2, $3
add  $1, $1, $4
sub  $5, $5, $1
```

Register $1 is written by *both* preceding instructions; only the most recent (from the 2nd ADD) is to be forwarded

add $1, $2, $3

add $1, $1, $4

sub $5, $5, $1



8

MEM/WB hazard equations

Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.

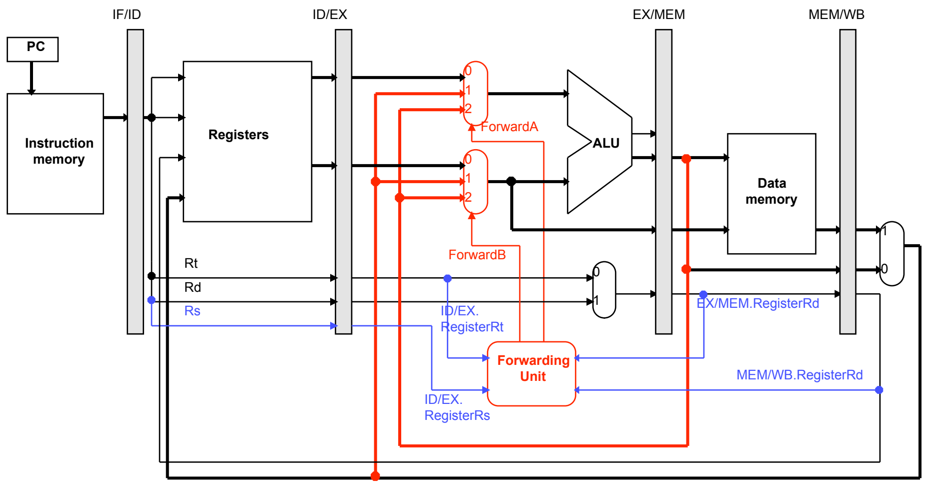
```
if (MEM/WB.RegWrite == 1
   && MEM/WB.RegisterRd == ID/EX.RegisterRs
   && (EX/MEM.RegisterRd != ID/EX.RegisterRs ||
         EX/MEM.RegWrite == 0)
then ForwardA = 1
```

The second ALU operand is handled similarly

```
if (MEM/WB.RegWrite == 1
   && MEM/WB.RegisterRd == ID/EX.RegisterRt
   && (EX/MEM.RegisterRd != ID/EX.RegisterRt ||
         EX/MEM.RegWrite = 0)
then ForwardB = 1
```

9

Simplified datapath with forwarding



10

The forwarding unit

The forwarding unit has several control signals as inputs

ID/EX.RegisterRs EX/MEM.RegisterRd MEM/WB.RegisterRd
ID/EX.RegisterRt EX/MEM.RegWrite MEM/WB.RegWrite

(The two RegWrite signals are not shown in the diagram, but they come from the control unit.)

The fowarding unit outputs are selectors for the ForwardA and ForwardB multiplexers attached to the ALU. These outputs are generated from the inputs using the equations on the previous pages

Some new buses route data from pipeline registers to the new muxes

11

Example

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

Assume again each register initially contains its number plus 100

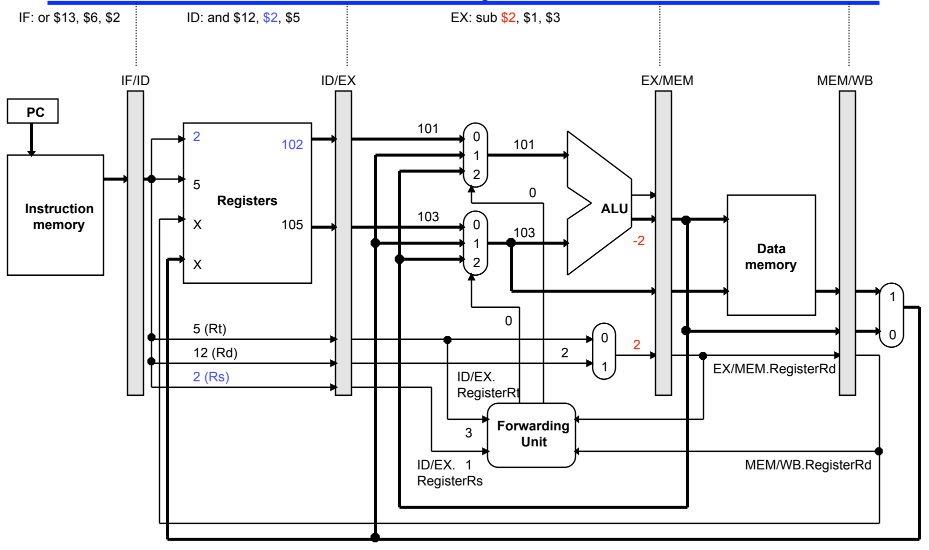
- After the first instruction, $2 should contain -2 (101 - 103)
- The other instructions should all use -2 as one of their operands

We'll try to keep the example short

- Assume no forwarding is needed except for register $2
- We'll skip the first two cycles, since they're the same as before

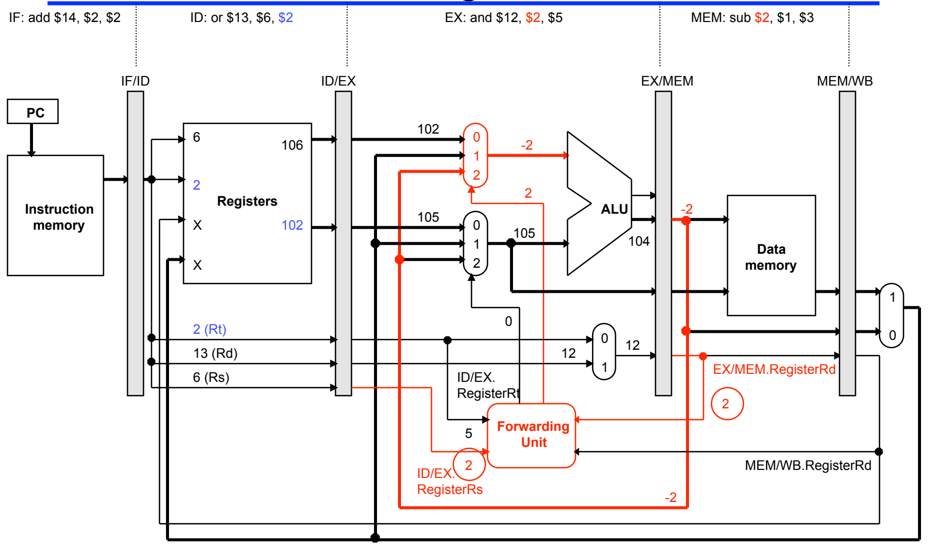
12

Clock cycle 3



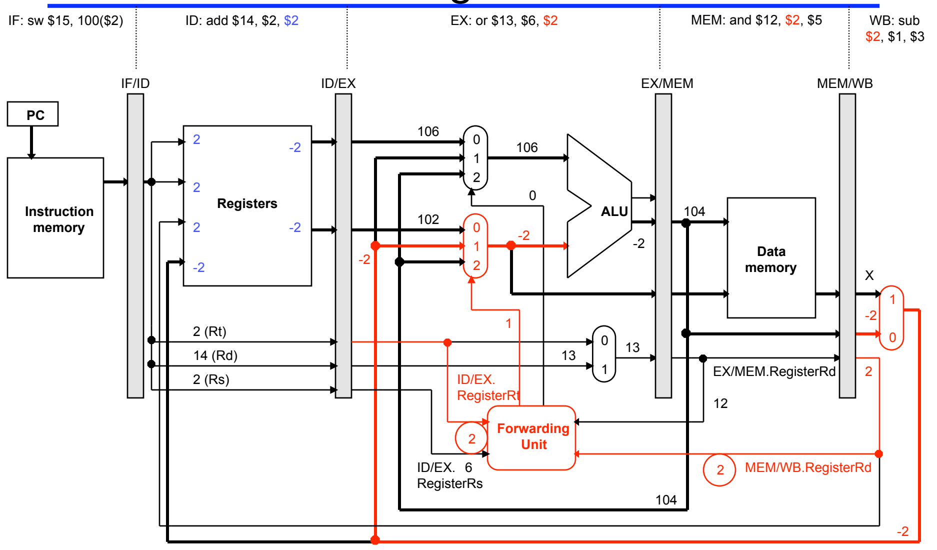
13

Clock 4: forwarding $2 from EX/MEM



14

Clock 5: forwarding $2 from MEM/WB



15

Lots of data hazards

The first data hazard occurs during cycle 4.

- ❖ The forwarding unit notices that the ALU’s first source register for the AND is also the destination of the SUB instruction.
- ❖ The correct value is forwarded from the EX/MEM register, overriding the incorrect old value still in the register file.

A second hazard occurs during clock cycle 5.

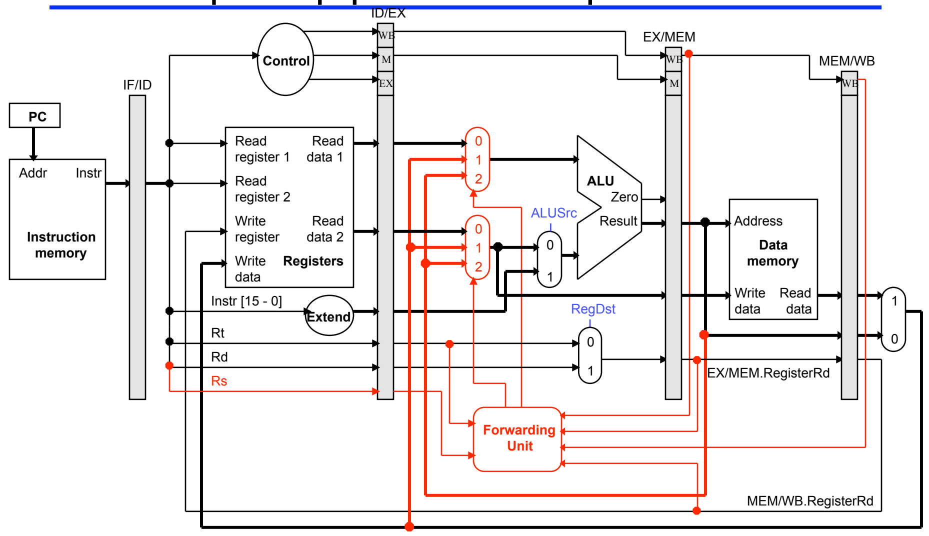
- ❖ The ALU’s second source (for OR) is the SUB destination again.
- ❖ This time, the value has to be forwarded from the MEM/WB pipeline register instead.

There are no other hazards involving the SUB instruction.

- ❖ During cycle 5, SUB writes its result back into register $2.
- ❖ The ADD instruction can read this new value from the register file in the same cycle.

16

Complete pipelined datapath...so far



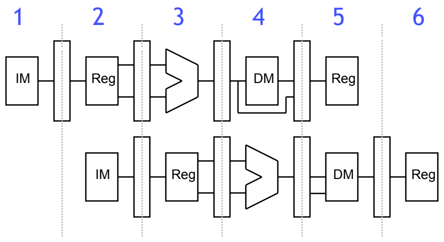
17

What about stores?

A harder case:

lw $1, 0($2)

sw $1, 0($4)



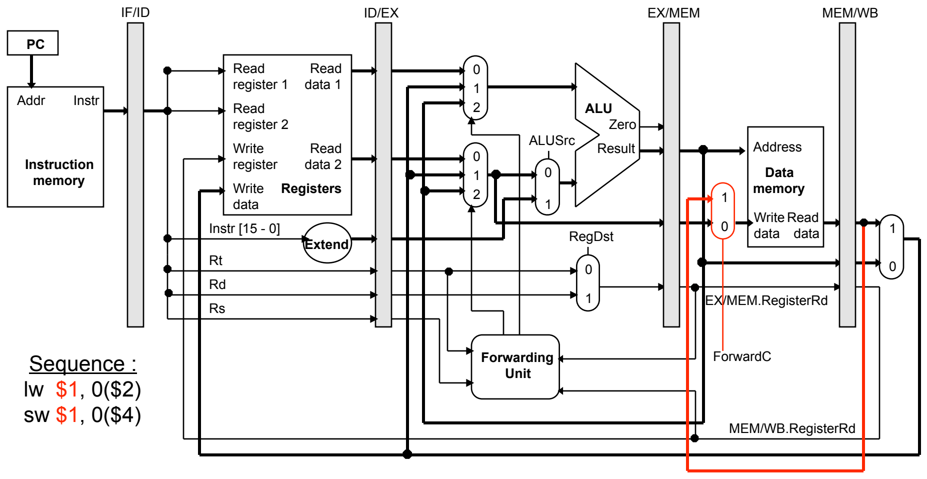
In what cycle is:

- The load value available?
- The store value needed?

What do we have to add to the datapath?

18

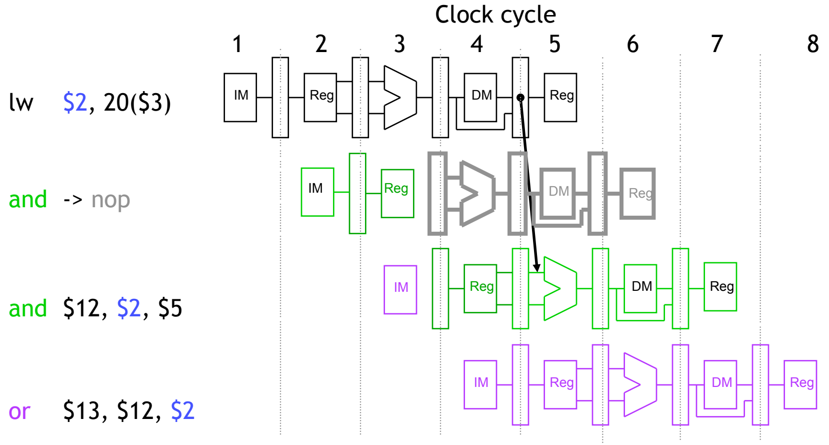
Load/Store Bypassing: Extends Datapath



Sequence :
lw $1, 0($2)
sw $1, 0($4)

19

Stall = Nop conversion



The effect of a load stall is to insert an empty or nop instruction into the pipeline

20

Detecting stalls

We can detect a load hazard between the current instruction in its ID stage and the previous instruction in the EX stage just like we detected data hazards

A hazard occurs if the previous instruction was LW...

ID/EX.MemRead == 1

...and the LW destination is a current source register

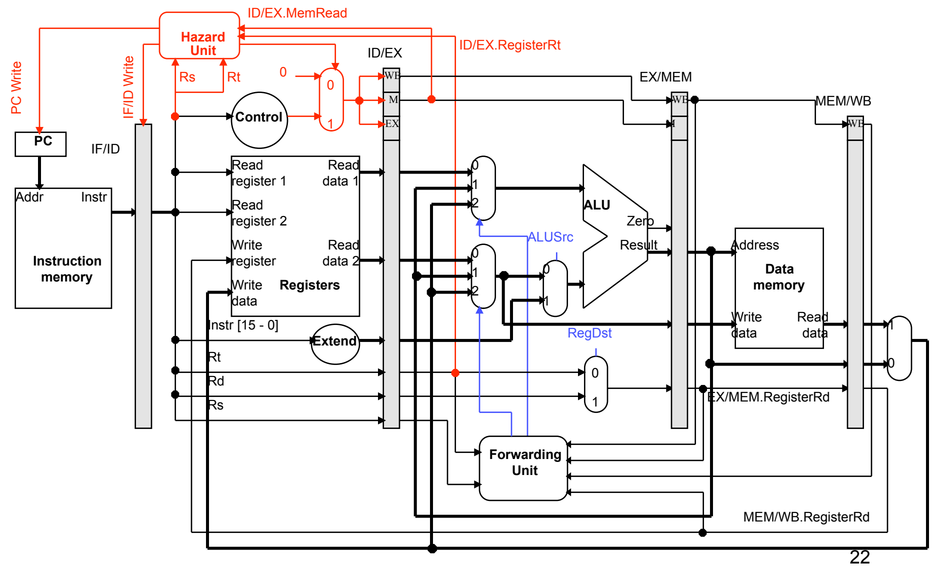
ID/EX.RegisterRt == IF/ID.RegisterRs

or

ID/EX.RegisterRt == IF/ID.RegisterRt

The complete test for stalling is the conjunction of these two conditions

Adding hazard detection to the CPU



The hazard detection unit

The hazard detection unit's inputs are as follows:

– IF/ID.RegisterRs and IF/ID.RegisterRt, the source registers

– ID/EX.MemRead and ID/EX.RegisterRt

By inspecting these values, the detection unit generates three outputs

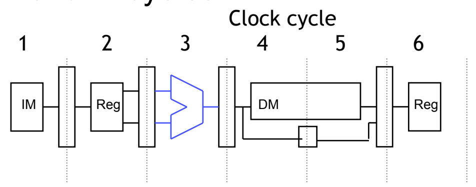
❖ Two new control signals PCWrite and IF/ID Write, which determine whether the pipeline stalls or continues

❖ A mux select for a new multiplexer, which forces control signals for the current EX and future MEM/WB stages to 0 in case of a stall

23

Generalizing Forwarding/Stalling

What if data memory access was so slow, we wanted to pipeline it over 2 cycles?



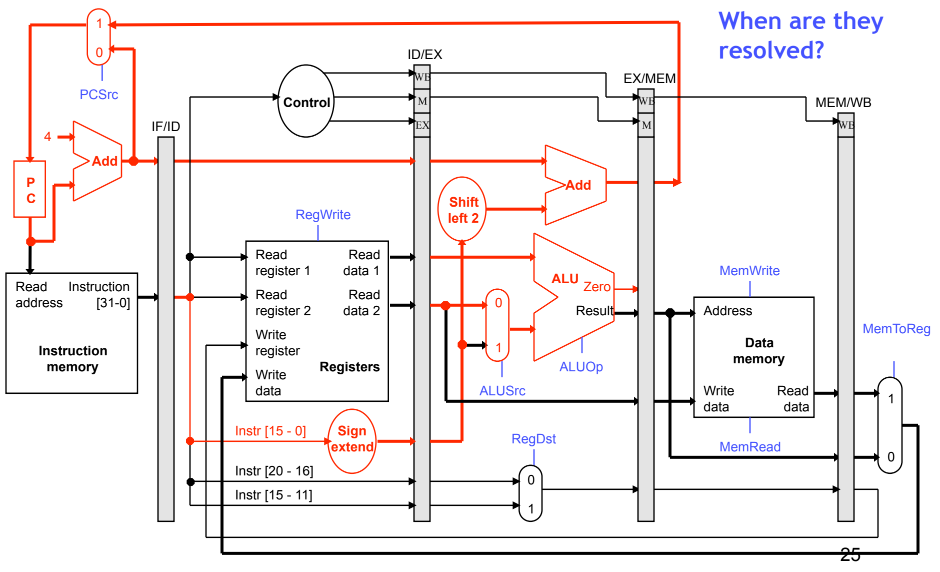
How many bypass inputs would the muxes in EX have?

Which instructions in the following require stalling and/or bypassing?

```
lw      r13, 0(r11)
add     r7, r8, r9
add     r15, r7, r13
```

24

Branches in the original datapath



When are they resolved?

25

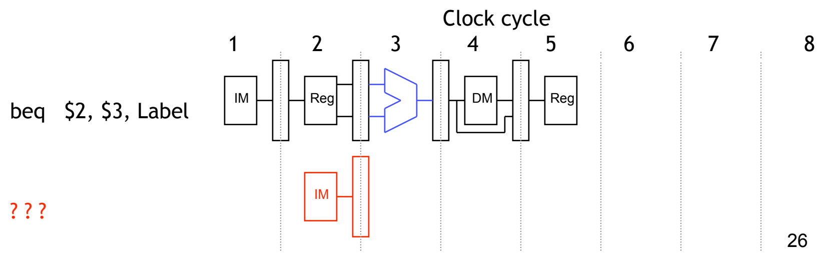
Branches

Most of a branch computation is done in the EX stage

- ❖ The branch target address is computed
- ❖ The 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

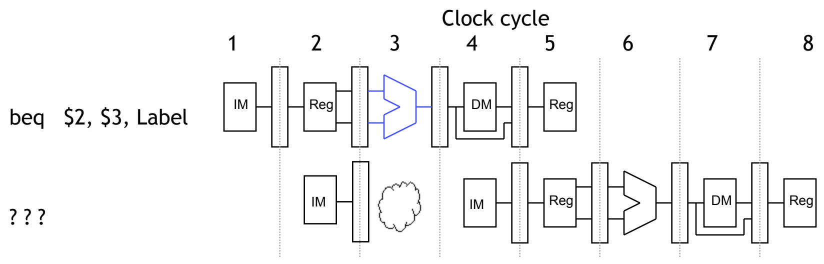
- ❖ But we need to know which instruction to fetch next, in order to keep the pipeline running!
- ❖ This leads to what's called a control hazard



26

Stalling is one solution

Again, stalling is always one possible solution



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

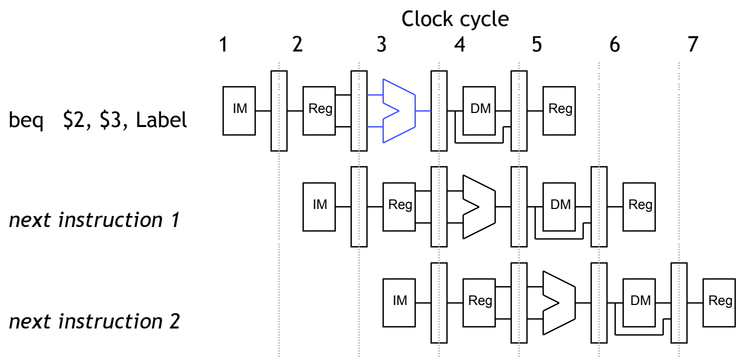
27

Branch prediction

Another approach is to *guess* whether branch is taken

- ❖ For 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.

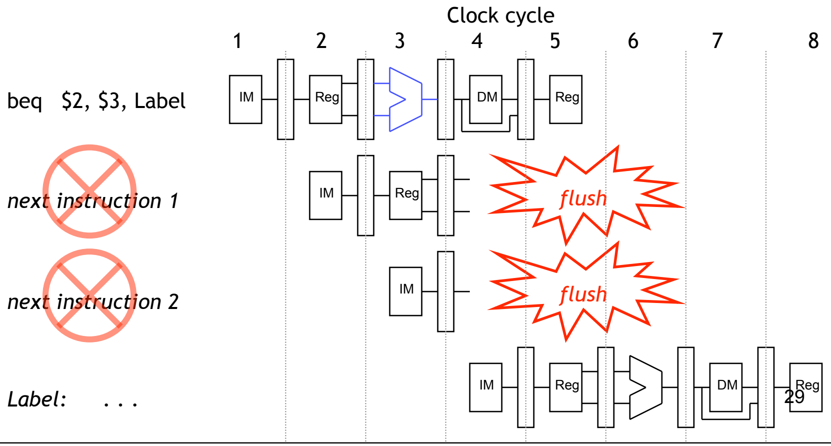
If we're correct, then the pipeline runs at full speed



28

Branch misprediction

If we guess wrong, then we have already started executing 2 instructions incorrectly and must discard, or flush, those instructions and begin executing the right ones from the branch target address, Label



29

Performance gains and losses

Overall, branch prediction is worth it

- ❖ Mispredicting a branch wastes two clock cycles
- ❖ But if the prediction is even occasionally correct, then it is preferable to stalling and wasting 2 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 run-time to determine the likelihood of a branch being taken

Pipeline structure also has a big impact on branch prediction

- ❖ A deeper pipeline may require more instructions be flushed on a misprediction, resulting in more wasted time, lower performance
- ❖ We must also be careful that instructions do not modify registers or memory before they get flushed

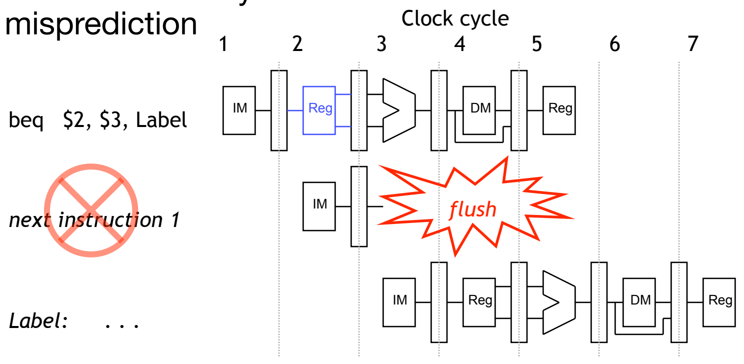
30

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



31

Implementing flushes

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

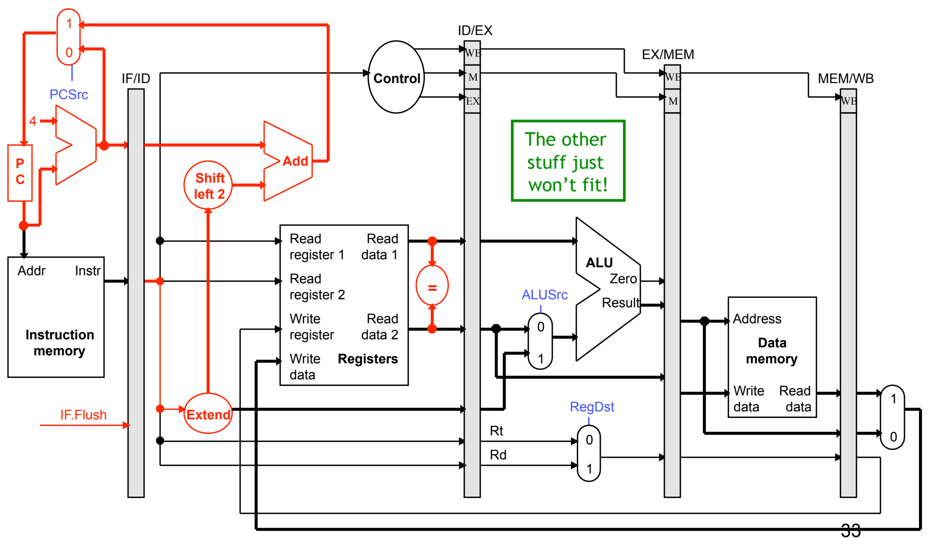
We can flush an instruction from the IF stage by replacing it in the IF/ID pipeline register with a harmless nop instruction

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

32

Branching *w/o* forwarding and load stalls



33

Timing

If no prediction:

IF ID EX MEM WB
IF IF ID EX MEM WB --- lost 1 cycle

If prediction:

- ❖ If Correct
  - IF ID EX MEM WB
  - IF ID EX MEM WB -- no cycle lost
- ❖ If Misprediction:
  - IF ID EX MEM WB
  - IF0 IF1 ID EX MEM WB --- 1 cycle lost

34

Summary

Three kinds of hazards make pipelining difficult

Structural hazards result from not having enough hardware to execute multiple instructions at once

- ❖ These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline.

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 pipe

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

35