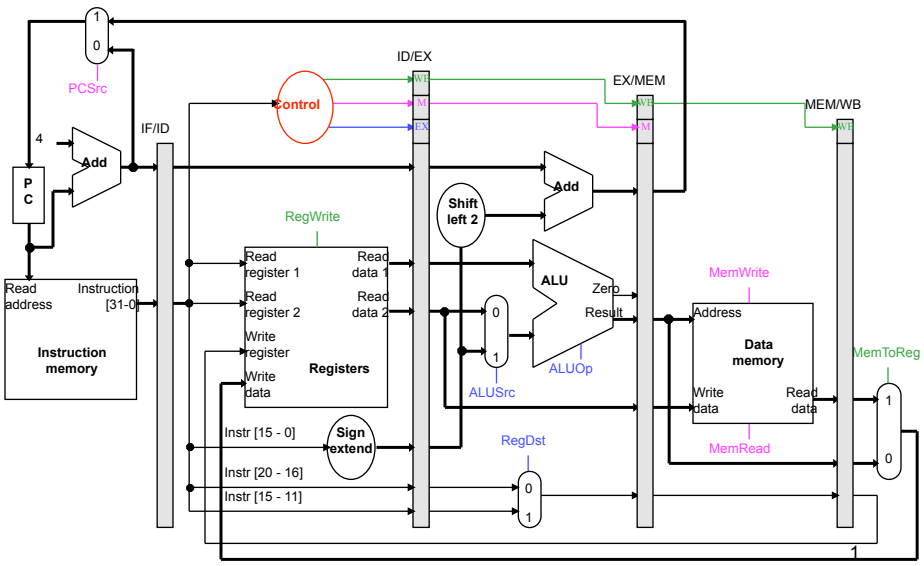
Pipeline Review



Our examples are too simple

Here is the example instruction sequence used to illustrate pipelining on the previous page

```
lw   $8, 4($29)
sub  $2, $4, $5
and  $9, $10, $11
or   $16, $17, $18
add  $13, $14, $0
```

The instructions in this example are independent

- Each instruction reads and writes completely different registers
- Our datapath handles this sequence easily

But most sequences of instructions are *not* independent!

2

An example with dependences

Read after Write dependences

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

Dependences are a property of how the computation is expressed

3

An example with dependences

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

There are several dependences in this code fragment

- The first instruction, SUB, stores a value into $2
- That register is used as a source in the rest of the instructions

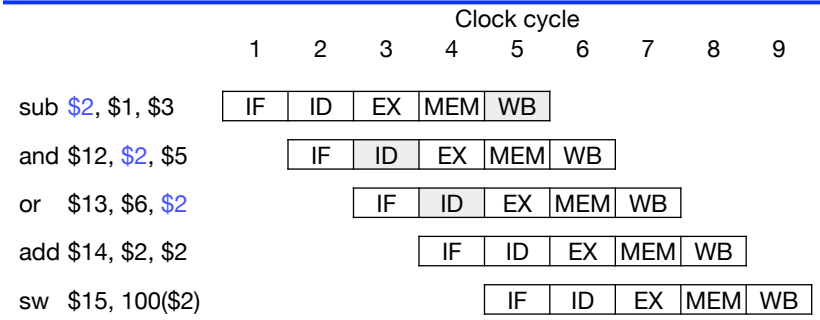
This is no problem for 1-cycle and multicycle datapaths

- Each instruction executes completely before the next begins
- This ensures that instructions 2 through 5 above use the new value of $2 (the sub result), just as we expect.

How would this code sequence fare in our pipelined datapath?

4

Data hazards in the pipeline diagram

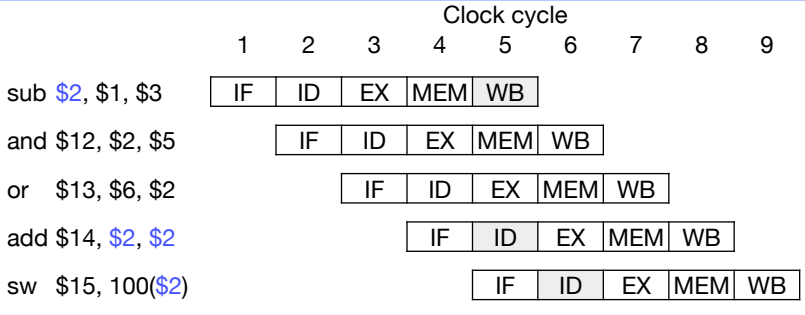


The SUB does not write to register $2 until clock cycle 5 causeing 2 data hazards in our pipelined datapath

- The AND reads register $2 in cycle 3. Since SUB hasn't modified the register yet, this is the *old* value of $2
- Similarly, the OR instruction uses register $2 in cycle 4, again before it's actually updated by SUB

5

Things that are okay



The ADD is okay, because of the register file design

- Registers are written at the beginning of a clock cycle
- The new value will be available by the end of that cycle

The SW is no problem at all, since it reads $2 after the SUB finishes

6

One Solution To Data Hazards

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

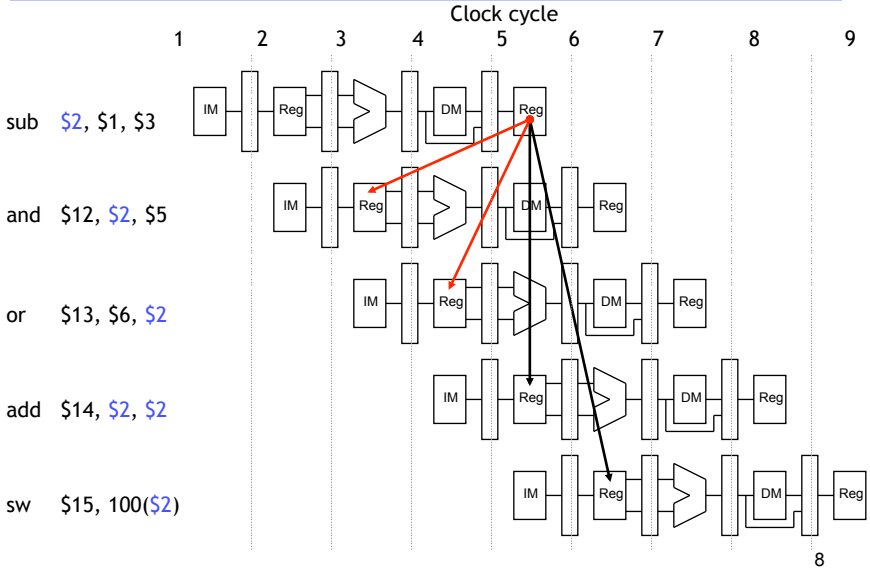
```
sub  $2, $1, $3
sll  $0, $0, $0
sll  $0, $0, $0
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

Since it takes two instruction cycles to get the value stored, one solution is for the assembler to insert no-ops or for compilers to reorder instructions to do useful work while the pipeline proceeds

A software solution to data hazards

7

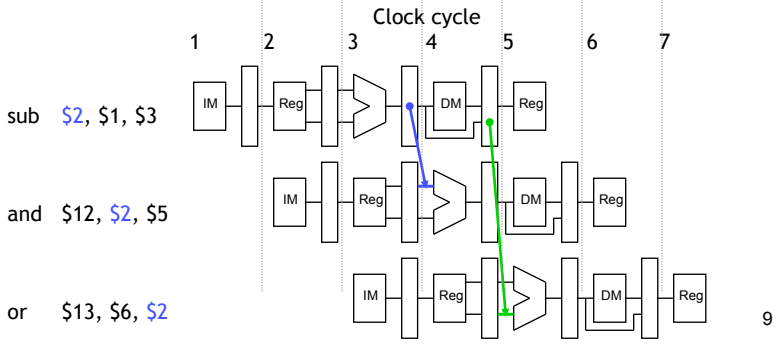
A fancier pipeline diagram



Forwarding

Since the pipeline registers already contain the ALU result, we could just forward the value to later instructions, to prevent data hazards

- In clock cycle 4, the AND instruction can get the value of $1 – $3 from the EX/MEM pipeline register used by SUB
- Then in cycle 5, the OR can get that same result from the MEM/WB pipeline register being used by SUB



9

Forwarding Implementation

Forwarding requires …

(a) Recognizing when a potential data hazard exists, and

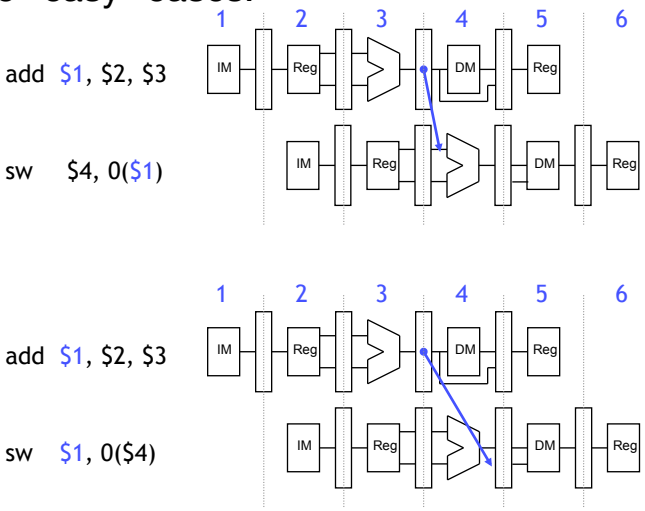
(b) Revising the pipeline to introduce forwarding paths …

We'll do those revisions next time

10

What about stores?

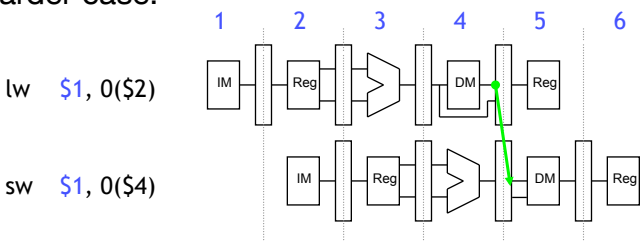
Two “easy” cases:



11

What about stores?

A harder case:



In what cycle is:

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

What do we have to add to the datapath?

12

Load/Store Bypassing: Extends Datapath

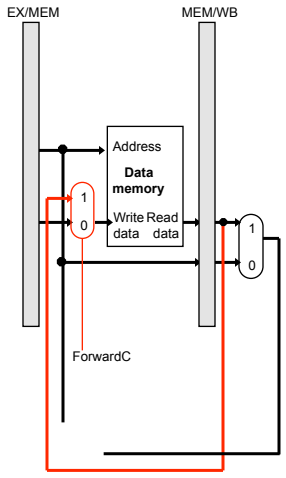
By cycling the result of Read data back to be the value for Write data, the combination

Sequence :

lw $1, 0($2)

sw $1, 0($4)

can operate at normal pipeline speeds … until there is a cache miss!



13

Stalls and flushes

We have seen data hazards can occur in pipelined CPUs when instructions depend upon others still executing

- Many hazards can be resolved by forwarding data from the pipeline registers, instead of waiting for the writeback stage
- The pipeline continues running at full speed, with one instruction beginning on every clock cycle

Now, we'll see some real limitations of pipelining

- Forwarding may not work for data hazards from load instructions
- Branches affect the instruction fetch for the next clock cycle

In both of these cases we may need to slow down, or stall, the pipeline

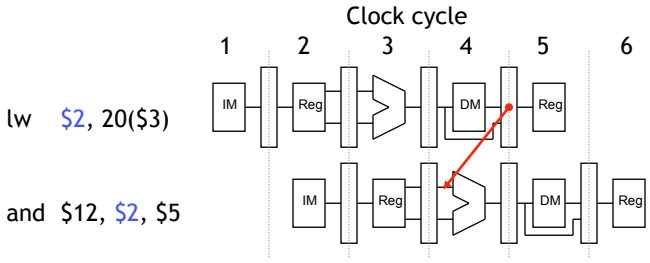
14

What about loads?

Imagine if the first instruction in the example was LW instead of SUB

- The load data doesn't come from memory until the *end* of cycle 4
- But the AND needs that value at the *beginning* of the same cycle!

This is a "true" data hazard—the data is simply not *available* when its needed

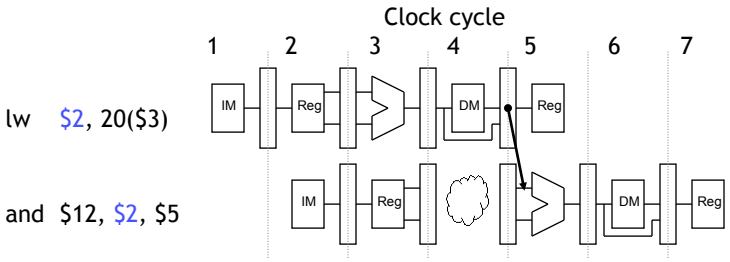


15

Stalling

The easiest solution is to stall the pipeline

We can delay the AND instruction by introducing a 1 cycle delay in the pipeline, often called a bubble

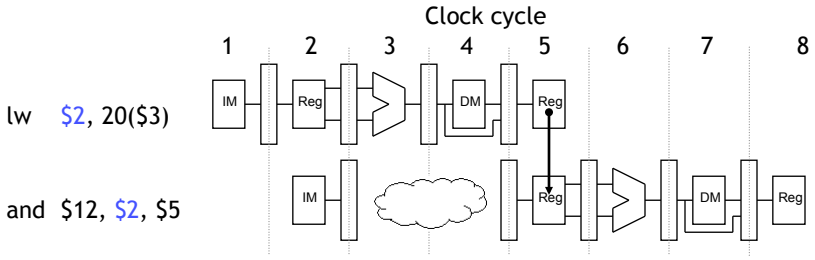


Notice that we're still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU

16

Stalling and forwarding

Without forwarding, we'd have to stall for *two* cycles to wait for the LW instruction's writeback stage.



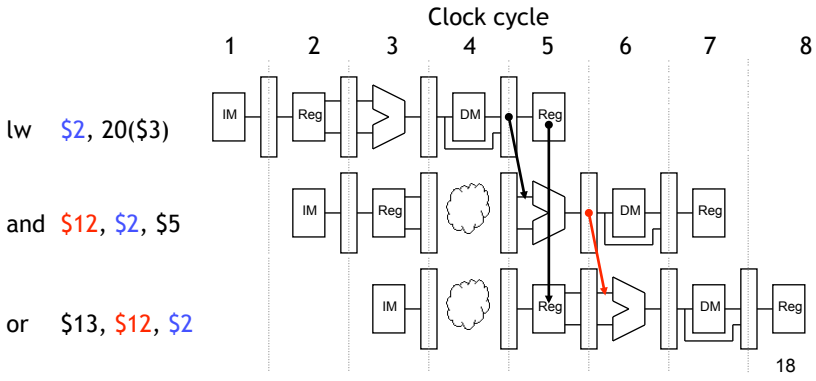
In general, you can always stall to avoid hazards—but dependencies are very common in real code, and stalling will often reduce performance significantly

17

Stalling delays the entire pipeline

If we delay the 2nd instruction, we must delay the 3rd too

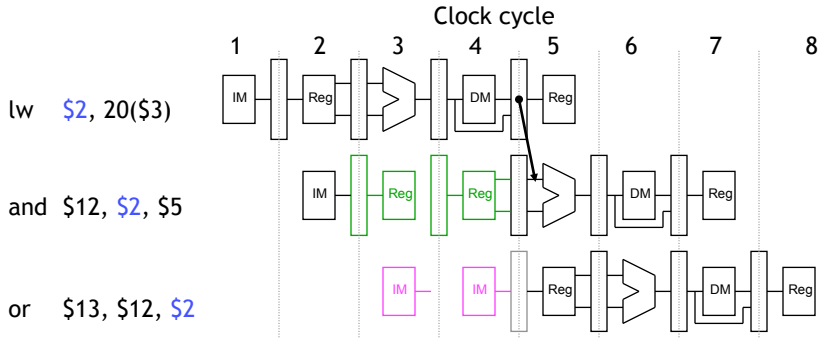
- This is necessary to make forwarding work between AND and OR
- It also prevents problems such as two instructions trying to write to the same register in the same cycle.



18

Implementing stalls

To implement a stall we force the two instructions after LW to remain in their ID & IF stages for 1 extra cycle



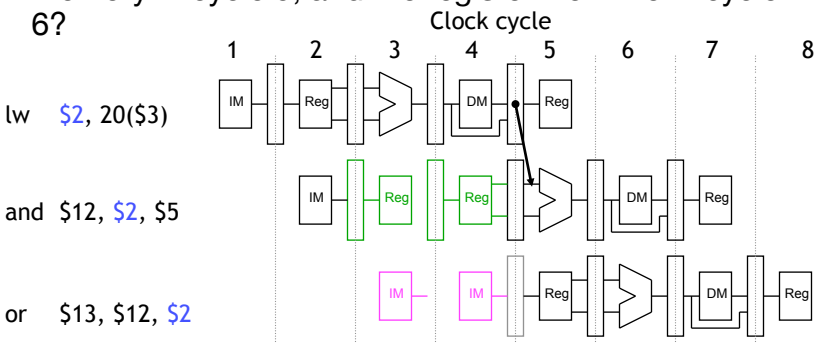
This is easily accomplished

- Don't update the IF/ID register, so the ID stage is repeated
- Don't update the PC, so the current IF stage is repeated

19

What about EXE, MEM, WB

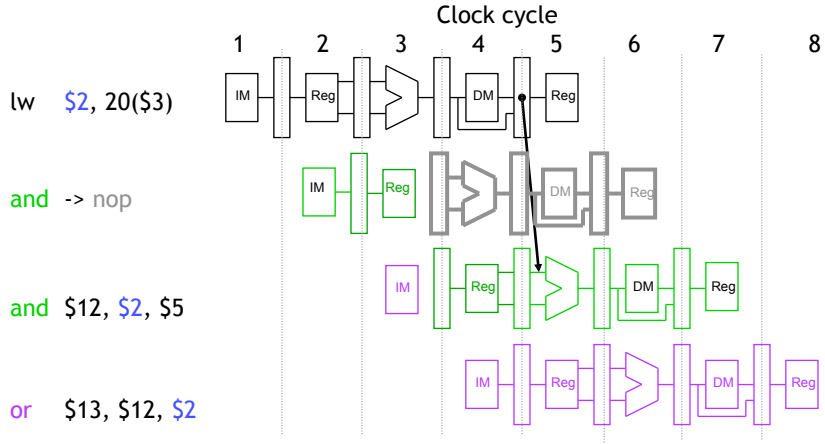
But what about the ALU during cycle 4, the data memory in cycle 5, and the register file write in cycle 6?



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 … the bubble "bubbles" through

20

Stall = Nop conversion



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

21