Data Hazards

[H&P §3.1] In the past several lectures, we have been discussing control hazards. Earlier, we discussed structural hazards briefly.

The third kind of hazard is data hazards. Data hazards come in three varieties.

Let *i* and *j* be two instructions in the same program, such that *j* follows *i*.

- Read-after-write (RAW). If an item is read by *j before* it is written by *i*, an error occurs ("read too soon").
- Write-after-read (WAR). If an item is written by *j before* it is read by *i*, an error occurs ("written too soon").
- Write-after-write (WAW). If *j* writes an item before *i* writes it, an error occurs ("written out-of-order").

Example: Consider this program. All the "variables" refer to registers, except for the array *M*, which is main memory.

1. $ac \leftarrow ac + M[1024]$;
2. $M[1000] \leftarrow sp + 1$;
3. $M[1000] \leftarrow ac$;
4. $a \leftarrow a - M[1000]$;

There are four hazards in this code fragment. What are they?

- The first is
- The second is
- The third is
- The fourth is

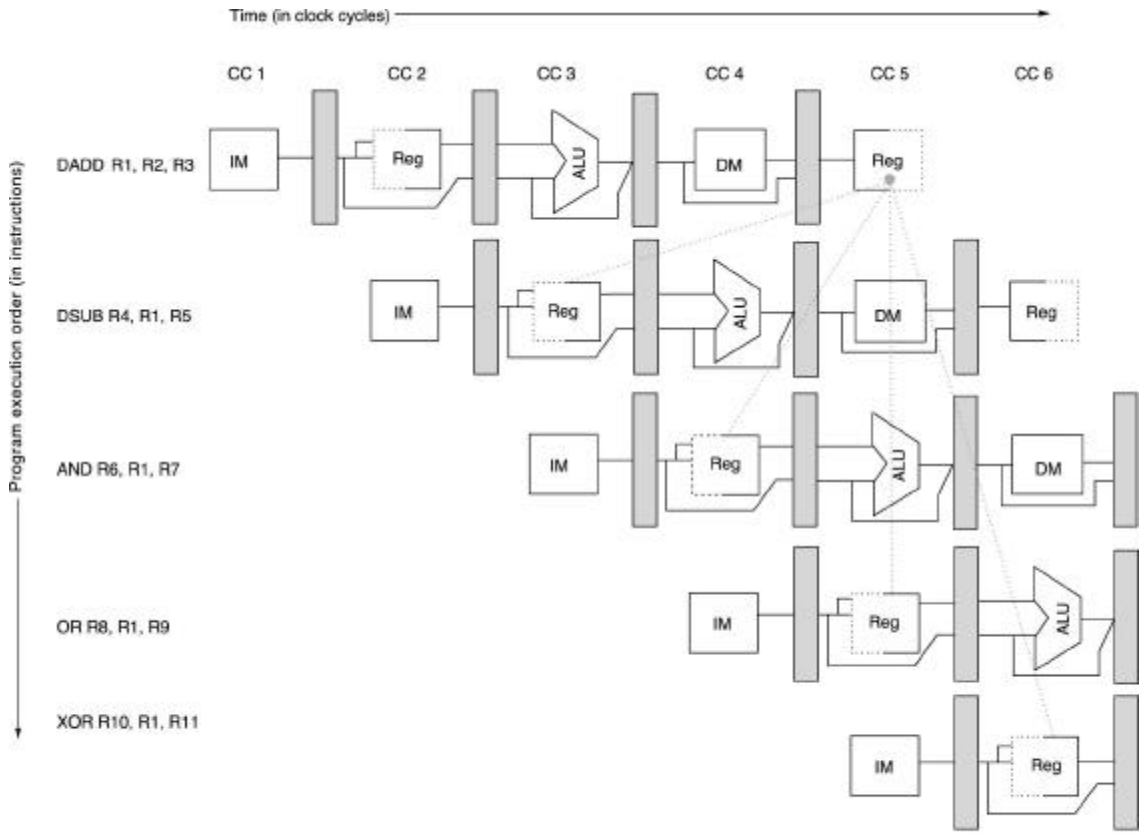
Just like control hazards, data hazards often cause stalls, and there are techniques to minimize these stalls.

Data dependences are properties of the code, but the presence of a hazard and the length of any stall is a property of the pipeline.

Read-after-write (RAW) hazard

RAW hazards are the most common kind.

Here is an example of a data hazard, from §A.2 of H&P.



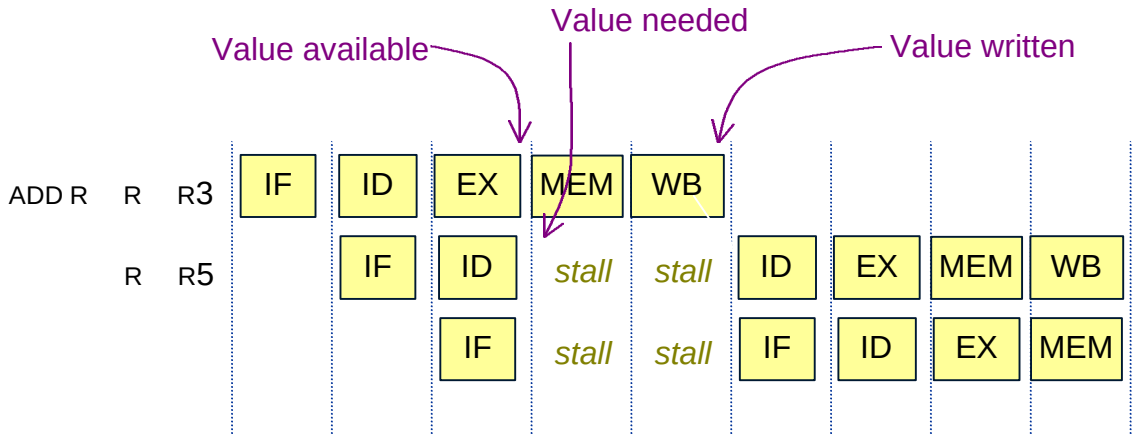
All the instructions after the DADD use the result of the DADD instruction.

Unless something is done to prevent it, the next three instructions may receive the wrong value.

In fact, due to the possibility of an interrupt, the result is not even deterministic.

Program order must be preserved to assure that *j* receives the value from *i*.

The easiest way to do that is to stall a later instruction until the value is available:



Data forwarding (bypasses)

Stalls can often by eliminated by *bypasses*, or *data forwarding*.

Notice in the diagram above that the value is not actually needed until after it is computed.

If the result can be moved

- from the pipeline register where the first instruction's EX stage produces it
- directly to the input of the next instruction's EX stage,

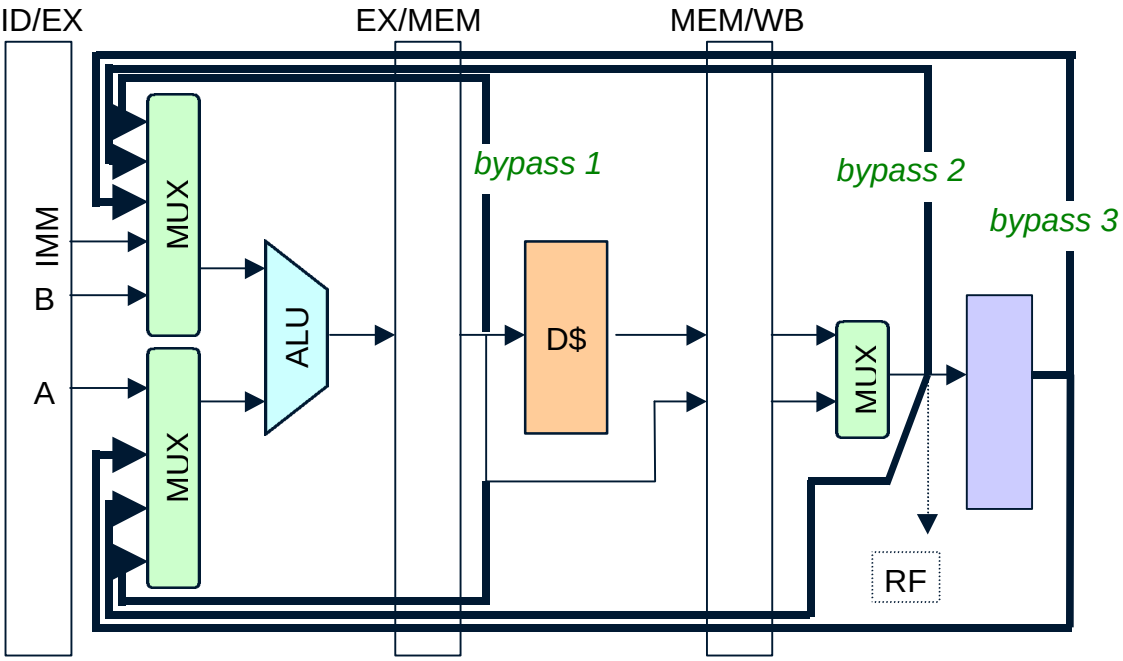
then no stall will be necessary.

In this pipeline, three bypasses can be built in, to allow the following three instructions to complete without stalls:

Clock #	1	2	3	4	5	6	7	8	9
ADD R1, R2, R3	IF	ID	EX	MEM	WB				
ADD R4, R1, R5		IF	ID	EX	MEM	WB			
ADD R6, R1, R5			IF	ID	EX	MEM	WB		
ADD R7, R1, R5				IF	ID	EX	MEM	WB	
ADD R8, R1, R5					IF	ID	EX	MEM	WB

byp 1 byp 2 byp 3

Bypasses can be implemented as data paths connecting the outputs of later stages to the inputs of earlier stages:



Stalls due to data hazards

In our simple pipeline

- Most RAW hazards do not cause a stall.
- Loads cause 1-cycle stall (in the case of a cache hit).

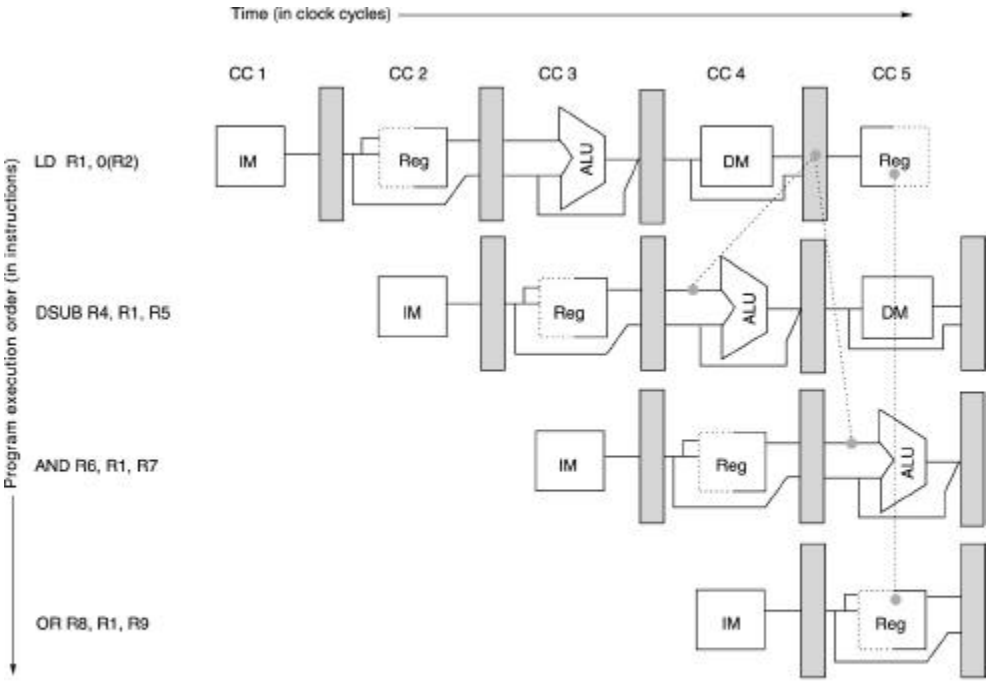
Essentially, this is because the “result” of a load is produced in the MEM stage, not the EX stage:

Value needed *Value available*

Clock #	1	2	3	4	5	6	7	8	9
LW **R1**, 0(R2)	IF	ID	EX	MEM	WB				
ADD R4, **R1**, R5		IF	*ID*	*stall*	EX	MEM	WB		
ADD R6, **R1**, R5			*IF*	*stall*	ID	EX	MEM	WB	

byp 2

This example from the text shows how the results of a load can be forwarded to instructions beginning two or more cycles later:



For the case where a stall is needed, we need a *pipeline interlock*.

A pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared.

For the above situation, where would stalls need to be inserted?

Clock #	1	2	3	4	5	6	7	8	9
LW **R1**, 0(R2)	IF	ID	EX	MEM	WB				
DSUB R4, **R1**, R5									
AND R6, **R1**, R7									
OR R8, **R1**, R9									

Which bypasses are used in this example?

Write-after-read (WAR) hazard

A WAR hazard occurs when a value is "written too soon."

The following code may produce a WAR hazard. How?

```
AND   R1, R2, R3
OR    R2, R4, R5
```

Can WAR hazards happen in our simple 5-stage pipeline?

How do you know this?

WAR hazards can happen …

- if some instructions write registers _____ and other instructions _____, or
- when instructions are reordered.

Here is an example of code that produces a WAR hazard.

Clock #	1	2	3	4	5	6	7
SW 0(R2), **R1**	IF	ID	EX	MEM1	MEM2	MEM3	WB
ADD **R1**, R3, R4		IF	ID	EX	WB		

Writes R1 during WB *Reads R1 during MEM3*

Write-after-write (WAW) hazard

A WAW hazard occurs when two values are "written out of order."

The following code may produce a WAW hazard.

```
AND  R1, R2, R3
OR   R1, R4, R5
```

The result is that later instructions see the wrong value in the register.

Can WAW hazards happen in our simple 5-stage pipeline?

How do you know this?

WAW hazards are present only in pipelines that

- write (to the same location) in more than one stage, or
- allow an instruction to proceed when a previous instruction is stalled.

Clock #	1	2	3	4	5	6
LW **R1**, 0(R2)	IF	ID	EX	MEM1	MEM2	WB
ADD **R1**, R2, R3		IF	ID	EX	WB	

Writes 2nd version of R1 *Writes 1st version of R1*

This can occur when some pipelines (e.g., integer) are short and others (e.g., floating point) are long.

Handling WAR/WAW hazards

- Stall the later instruction (in the ____ stage)
- Through the compiler:
- By hardware, using register renaming (we will discuss this under the next major topic – ILP).

Types of dependences

[H&P §3.1] If two instructions are *independent*, they may be executed concurrently in a pipeline without causing any stalls (assuming no structural hazards exist).

If two instructions are *dependent*, they must be executed in order (though they may be partially overlapped).

There are three kinds of dependences:

- Data dependences (true dependences, flow dependences)
- Name dependences
- Control dependences

Data dependences

An instruction *j* is data dependent on an instruction *i* iff—

- instruction *i* produces a result that may be used by instruction *j*, or
- instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *i*.

For example, what are the data dependencies in this code?

```
ADD  R1, R2, R3
SUB  R5, R3, R1
MUL  R4, R2, R5
```

What kind of hazards can be caused by data dependences?

Name dependences

A *name dependence* occurs when two instructions use the same register or memory location (a “name”), but there is no flow of data between the instructions associagted with that name.

Suppose that instruction *i* precedes instruction *j* in program order.

- An *antidependence* between *i* and *j* occurs when instruction *j* writes a location that *i* reads.
- An *output dependence* occurs when instruction *i* and *j* write the same location.

Example of antidependence:

```
ADD   R3, R2, R1
SUB   R1, R4, R5
```

What kind of hazards can be caused by antidependences?

How can antidependences be removed?

Example of output dependence:

```
ADD   R1, R2, R3
SUB   R1, R4, R5
```

What kind of hazards can be caused by output dependences?

How can output dependences be removed?

Both antidependences and output dependences are name dependences rather than data dependences, since there is no value being transmitted between the instructions.

The techniques for removing name dependences are collectively called *renaming*.

Renaming can either be done statically by a compiler or dynamically by the hardware.