Computer Architecture - CS 301

Rishit Saiya - 180010027, Assignment - 6

October 10, 2020

1

We have seen some software inspired methods to patch Data Hazards are as follows:

- Re-Ordering of Instructions so as to avoid Hazards.
- Adding **nop** instruction at specific order to avoid Hazards.

These methods reprimand the Hazard to a certain level but they have their own drawback as follows:

- Since Pipelining represents the internal structure of the Architecture, we would like expose less of its details to Compiler. Apart from the fact that the risk of higher complexity, the exposure of Pipelining details also increases the surface area of attack and hence can prove fatal in security perspective.
- Since the above methods have to performed by the compiler, we can't completely rely on its ability.
- There will always have to be a compiler compatible to a specific hardware and architecture and this will increase the man efforts and no generalisation would be made.

So, in order to increase efficiency and accuracy, we implement more of a hardware approach. In this question we will be discussing the **Interlock** approaches.

The following are 2 types of Interlocks in general: Data Locks, Branch Locks. Data Locks are used for Data Hazards and Branch Locks are used for Control Hazards. So we will elaborate on Data Locks here because we are asked regarding Data Hazards.

- ***Data Lock:*** It does not allows a consumer instruction to move beyond OF stage till it has fetches the correct values.

  We must understand that, using Data Lock Interlock gives the portability with variations in pipelines but our performance is compromised here in simple versions of

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
IF	**1**	**2**							
OF		**1**	**2**	**2**	**2**	**2**			
EX			**1**	**B**	**B**	**B**	**2**		
MA				**1**	**B**	**B**	**B**	**2**	
RW					**1**	**B**	**B**	**B**	**2**

Table 1: Pipeline - Data Lock Explanation

interlocking in pipelines.

The main task of the Data Lock is to stall the IF & OF stages. So basically, when an instruction reaches OF stage, it checks if it has a contradiction with any instructions in EX, MA, RW stages. If no contradiction, then normal flow would continue and if a contraction arises, then stall the pipeline (IF & OF stages).

We define a ***Bubble*** as a Nop instruction packet. This is used in implementation in Data Lock.

Let's consider an example as follows:

```
add r1, r2, r3
sub r4, r1, r2
```

Here, the instruction[1], r1 is assigned the value of r2 + r3 in the 5th Time Cycle and its value can be fetched only after 5th cycle by instruction[2]. So, till the 6th time cycle, instruction[2] has to be stalled to avoid hazards invasion and we do it using bubble.

In the Figure 1, the above instruction have been shown in Pipeline view. The rows are the instruction and the column are the time cycles. **B** is the Bubble.

Minimum number of Bubbles Required:
If 2 consecutive instructions say X & Y are having consecutive executions i.e., if instruction Y is causing a hazard to instruction X, then no.of bubbles have to be selected appropriately. Let's assume Instruction X is in x^{th} cycle in RW stage, then Instruction Y can fetch correct values only at or after $(x+1)^{th}$ cycle. So the minimum number of bubbles would be the number of bubbles required to stall Instruction Y at OF stage to avoid Data Hazard. Clearly if there is no conflict, then the no. of bubbles will be **0**.

When an instruction reaches the OF stage, we check if it has a conflict with any of the instructions in the EX, MA, and RW stages respectively. If it has conflict with the instruction at EX stage, then the no. of bubbles will be **3**. If it has conflict with the instruction at MA stage, then no. of bubbles will be **2** and the no. of bubbles will be 1 if its at RW stage. So, in case of conflict, the min no. of bubbles to be introduced would be **1**.

2

We have seen some software inspired methods to patch Data Hazards are as follows:

- Re-Ordering of Instructions so as to avoid Hazards.
- Adding **nop** instruction at specific order to avoid Hazards.

These methods reprimand the Hazard to a certain level but they have their own drawback as follows:

- Since Pipelining represents the internal structure of the Architecture, we would like expose less of its details to Compiler. Apart from the fact that the risk of higher complexity, the exposure of Pipelining details also increases the surface area of attack and hence can prove fatal in security perspective.
- Since the above methods have to performed by the compiler, we can't completely rely on its ability.
- There will always have to be a compiler compatible to a specific hardware and architecture and this will increase the man efforts and no generalisation would be made.

So, in order to increase efficiency and accuracy, we implement more of a hardware approach. In this question we will be discussing the **Forwarding** approaches.

The following paths are created following above rules: RW $\rightarrow$ OF, RW $\rightarrow$ EX, RW $\rightarrow$ MA (load to store) and MA $\rightarrow$ EX (ALU instructions, load, store).

Let's consider an example as follows:

```
add r1, r2, r3
sub r4, r1, r2
```

We know that Figure 1 shows the correct pipelining data, and we finally need the correct data fetch for r1 without any Hazard intervention. So, to be more efficient we forward values in above stages.

Let's consider a hazard pipeline to understand better as shown in Figure 2. In Figure 2, we see that in time cycle 4, Instruction[1] reaches MA stage and correct value of r1 has been

	1	2	3	4	5	6
IF	**1**	**2**				
OF		**1**	**2**			
EX			**1**	**2**		
MA				**1**	**2**	
RW					**1**	**2**

Table 2: Pipeline - Forwarding Explanation

	1	2	3	4	5	6	7	8
IF	**1**	**2**						
OF		**1**	**2**					
EX			**1**	**2**	**2**	**2**		
MA				**1**	**2 — B**	**B**	**2**	
RW					**1**	**2 — B**	**B**	**2**

Table 3: Pipeline - Load Use Hazard

reached. At the same time, instruction[2] is in EX stage and needs correct r1 value. So we **forward** r1 value from MA stage to EX stage. This process is defined as Forwarding.

Clearly, this process doesn't require stalling and hence doesn't require any bubble.

There is an exception, when there is a load instruction involved. Let's consider the following example.

```
ld r1, 5[r2]
mul r3, r1, r4
```

Why is this case (Load Use Hazard) unique? It is because value of r1 is written in time cycle 5 and as mentioned above instruction[2] requires correct value of r1 at the same time and clearly this is not possible using Forwarding. So we need to tweak the idea of Forwarding in this unique case by introducing bubbles here. Figure 3 explains the pipelining in that case.

The forwarding here is taking place from RW instruction[1] to EX instruction[2].

Minimum number of Bubbles Required:
As mentioned above, it depends on how conflict is arising in the instructions. In this case, the minimum number of bubbles to be introduced will be **0**, because in forwarding we connect wires to forward data to different stage to improve performance index. In the case of load-use where we need to introduce a bubble, the number of bubbles needed is **1**.