Q1. Discuss the following hazards

1. RAW hazard

The read after write is true dependency. In RAW, if the instructions are executed in sequential order, then there is no problem. But if the instructions are executed in parallel, then there is a problem with the results.

Example:

I1: **R3** <-- R1 + R2

I2: R5 <-- R4 – **R3**

Before the updation of the I1 instruction in the memory element, instruction I2 fetched the unupdated operand R3 for execution. Here, I2 instruction execution is truly dependent on I1 instruction completion. Before I2 instruction write operation completion(updation), for I1 instruction execution operand, this is known as the Read after Write (RAW) hazard.

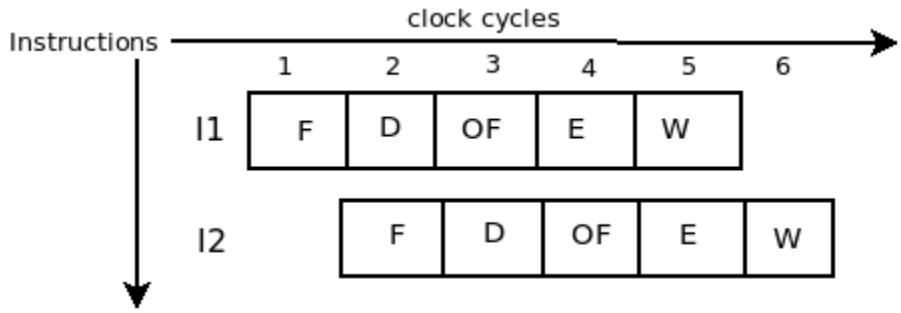

figure 1 RAW hazard

In figure 1, instruction I2 reads the operand before the I1 instruction write operation. So to avoid this problem, the solution is for instruction I2 operand to be fetched in the 6th clock cycle, which means after I1 instruction write operation completion.

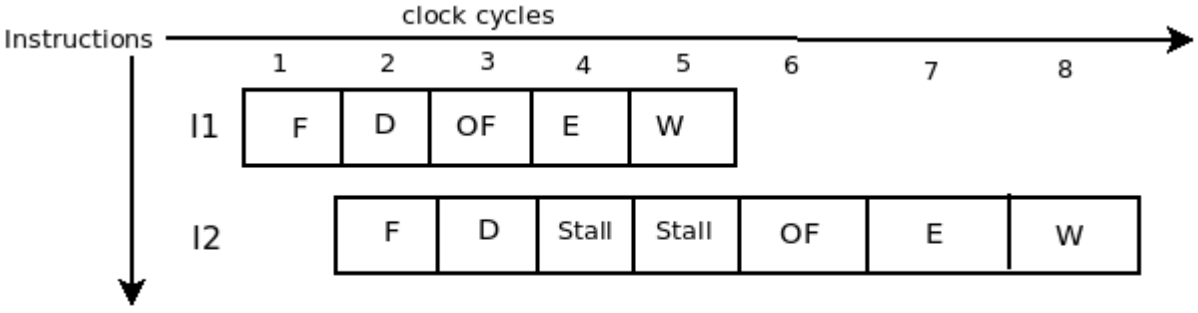

figure 2 RAW hazard solution

In Figure 1, for instruction I1, the R1 value is 20, the R2 value is 30, and the operation result is 50, which is stored in R3 and is updated in clock cycle 5. But if the I2 instruction requires an operand in the 4th clock cycle and the operand is fetched in the 4th clock cycle, then it is an unupdated operand. This is called data inconsistancy. To avoid this, in Figure 2, the I2 operand is fetched in the 6th clock cycle and the processor is idle in the 4th and 5th clock cycles. As a result,

the I2 instruction retrieved the updated operand (R3 value is 50), R4 value is 400, and the programme will produce correct results, which is R5 value is 350.

2. WAR hazard

A Write After Read hazard occurs when the instructions are executed concurrently. Suppose I1 and I2 are two instructions which are executed in a sequential order, then there is no problem. But in a special case, if I1 and I2 are executed in parallel, then there is a chance for WAR hazard.

Example

I1: R1 <-- R2 * **R3**
I2: **R3** <-- R4 + R5
I1: R1(200) <--R2(20) * R3(10)
I2: R3 (80) <-- R4 (30) + R5 (50)

In a four-or five-stage pipeline, there is a less chance of a "Write after Read" hazard. Suppose If I2 instruction execution is completed before I1 instruction operand fetch, then this WAR occurs. In the above example, if instructions are executed I2 after I1, then there is no WAR hazard occur. But instructions are executed after the completion of the I2 instruction write. If the I1 instruction reads the R3 updated value, then a WAR hazard occurs. The program will generate incorrect results, which is that the R1 value is 1600. Solution is register renaming.

I1: R1(1600) <-- R2(20) * R3(80)
I2: R3 (80) <-- R4 (30) * R5 (50)

3. WAW hazard

A Write After Write (WAW) hazard occurs if the instructions are executed concurrently. In the below given example, if two instructions, I1 and I2, were executed in parallel, it is assumed that Instruction I1 was delayed and that Instruction I2 was completed. In this situation, the WAW hazard occurs.

Example:

I1: **R3** <-- R1 + R2
I2: **R3** <-- R4 + R5

In the preceding example, R1 equals 50, R2 equals 50, R4 equals 100, and R5 equals 150. Suppose if instructions I1 and I2 are executed in a pipeline and after the completion of I1 instruction and I2 instruction, then R3 has the second instruction execution result. In this case, there is no problem. But the I1 instruction delayed execution, the I2 instruction completed its execution, and the R3(250) register holds the I2 instruction result. After some time, the I1 instruction also completed its execution and the result was overwritten in the R3(100) register. This is called the WAW hazard. Solution for WAW is destination register renaming.

I1: R3(100) <-- R1(50) + R2(50)
I2: R3(250) <-- R4(100) +R5(150)

Q2. What is branch prediction? The following branch prediction methods should be explainded.

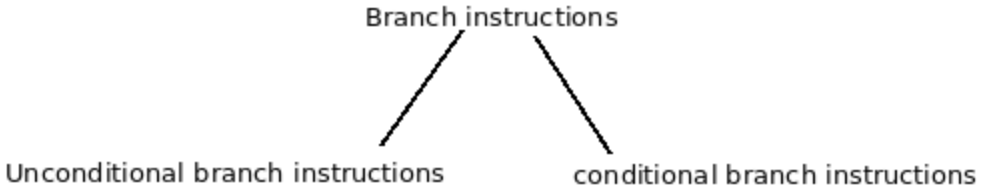

Figure 3 Branch Instructions

An unconditional branch instruction is nothing but a jump to a particular address without any condition check. Example: Jump 2000. Here 2000 is the instruction address location. The processor jumps to the 2000 address location and starts instruction execution.
If the condition is true, branch instructions are executed; otherwise, instructions are executed in the order they were received. example branch (if, if-else, loops).

Branch prediction

Hardware circuit tries to guesses whether a branch will be taken or not taken in the pipeline architecture. If a branch is detected in the pipeline while execution, then the processor waits for some clock cycles (stalls or bubbles). With this, pipeline performance is reduced. Branch is predicted before or during execution to improve pipelining performance.

1. static branch prediction

In static branch prediction, it is assumed that there is no branch in the pipeline and instructions are fetched in a sequential order. So if the instruction is detected, it means a branch in the pipeline stages. Then, whatever instructions are fetched in the pipeline, those are removed or flushed from the pipeline. After that, it jumps to the target branch and fetches instructions in a sequential order in the pipeline.

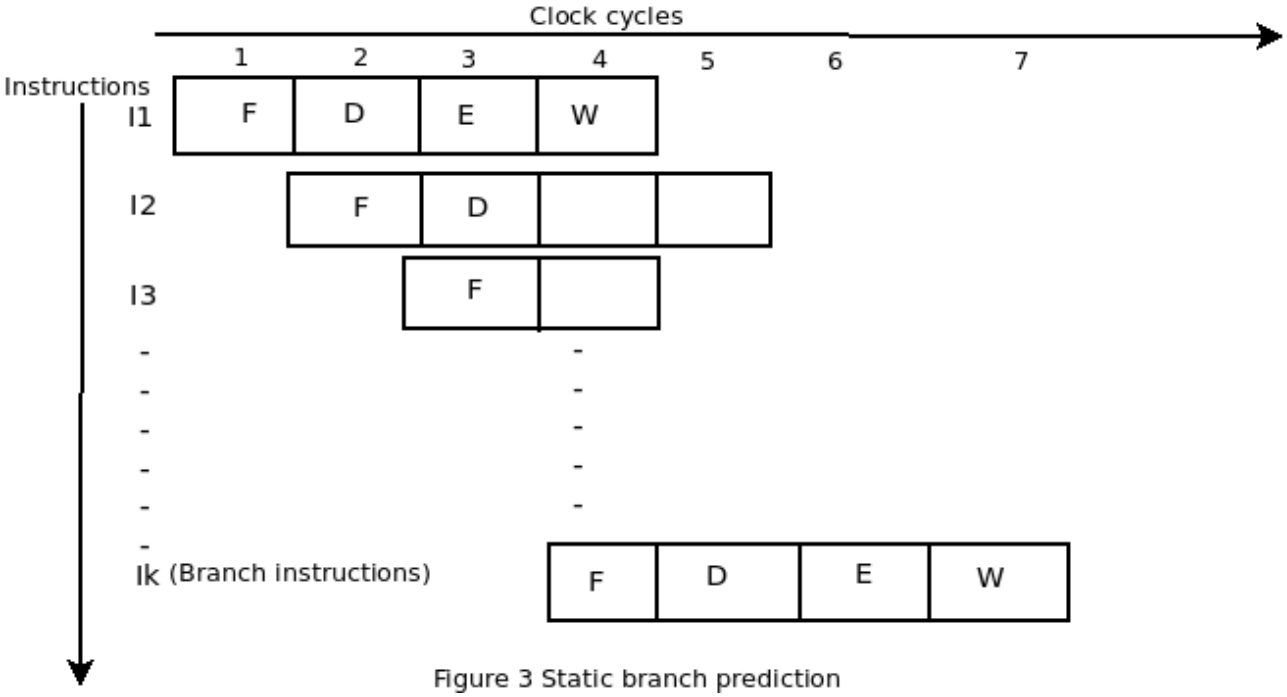

Figure 3 Static branch prediction

In Figure 3, I1, I2, and I3 instructions are in the pipeline in 3 clock cycles, and after the I2 instruction decode the processor knows branch has taken and instruction I3 is fetched in the pipeline. Before jumping to the target branch instruction, the I3 instruction was flushed(terminated) from the pipeline. So, in static branch prediction, instructions are fetched in the pipeline in a sequential order and, during execution branch has taken then jump to the target branch fetches instruction in sequential. If the branch has not taken, then instructions are fetched in sequential order.

Dynamic branch prediction

Dynamic branch prediction is a history-based prediction technique. It maintains a branch history table (BHT) that contains information about the previous branch.

2 state diagram (1 bit dynamic branch prediction)

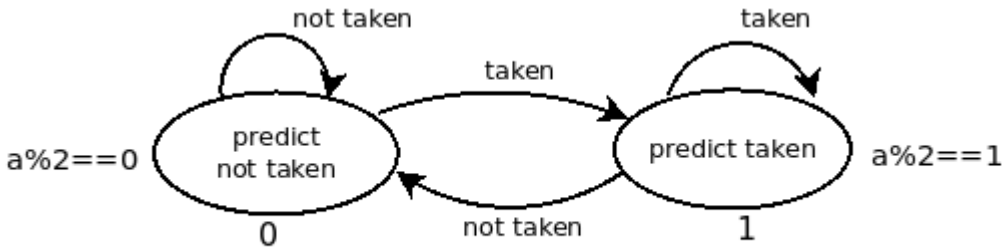

Figure 4 1 - bit dynamic branch prediction

In figure 4, there are two states predict not taken means binary value 0 and predict taken means binary value 1. The branch prediction table maintains previous branch information. Based on past information, it has been decided that either branch has taken or not taken.

Example:

```
int a=0;
while(a<5)
{
        if(a%2==0){
                branch instructions

        }
        a++;
}
```

- Branch to be taken. So at a=0 branch has taken
- At a=1 hardware assume that branch has taken but branch not taken
- At a=2 hardware assume that not taken but branch has to be taken
- At a=3 hardware assume that branch has taken but branch has not be taken
- At a=4 hardware assume that branch not taken but branch has to taken

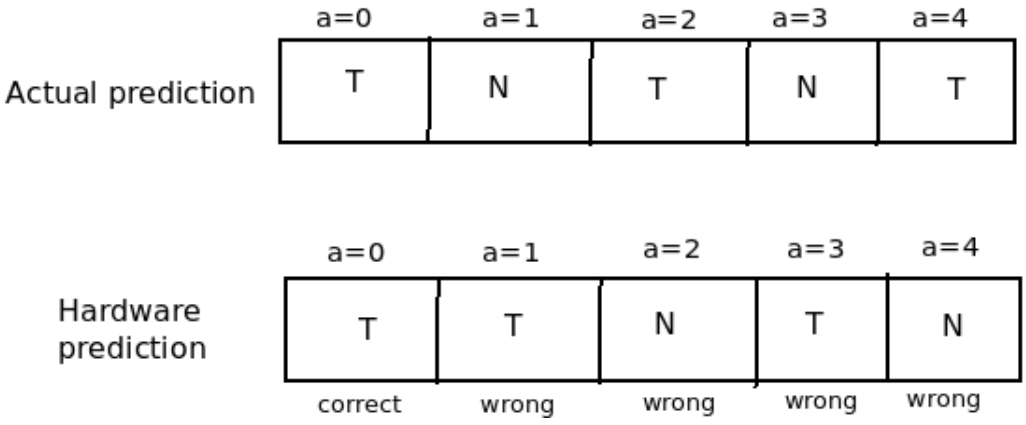

Figure 5 Actual prediction and Hardware prediction

In Figure 5, the branch has taken for at a=0, a=2, a=4 and the branch has not taken for at a=1, a=3 in actual prediction. However, because hardware predicts based on previous branch information, the branch at a=0 has taken. In actual prediction, at a=1 branch has not taken, but in hardware prediction, at a=1 branch has taken because previous history was branch taken. So if we

compared actual prediction with hardware prediction, only one case was correct at a=0 and all other cases were wrong.

4 state diagram (2 bit dynamic branch prediction)

A 2 bit dynamic branch prediction gives 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.

1 bit branch prediction	**2 bit branch prediction**
After one wrong branch prediction, prediction bit is inverted.	After two wrong predictions, then only prediction bit is inverted.

Example

```
for(int i=0; i<2;i++){
        for(int j=0;j<2;j++){
                ---
                branch instructions
        }
}
```

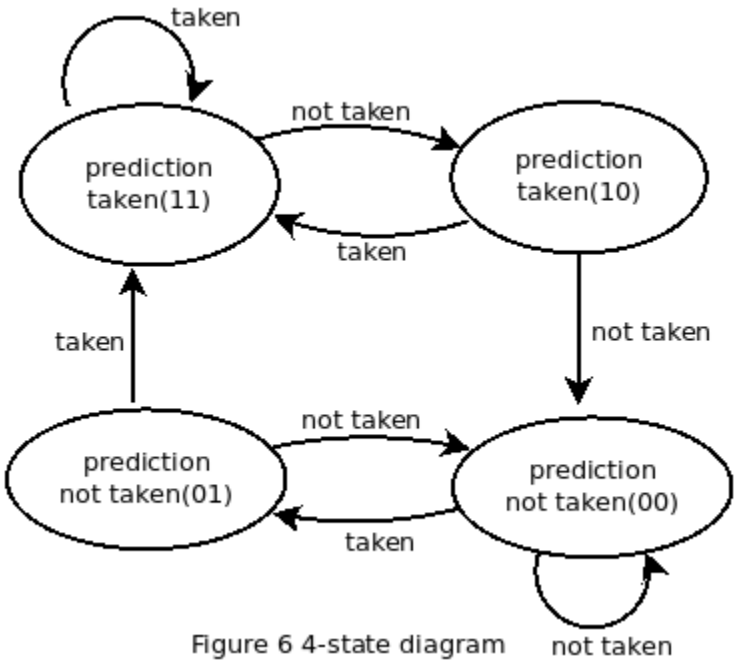

Figure 6 4-state diagram

Branch table initially contains 0 means not taken and 1 means taken. In above example loops for i=0 inner loop j repeats for two times. So, If initial is not taken then for next branch it will be considered as not taken only even it is taken.

	i=0	i=1	i=2
Actual prediction	J = 0, 1, 2 T T NT	J = 0, 1, 2 T T NT	J = 0, 1, 2 T T NT

	i=0	i=1	i=2
Hardware prediction	J = 0, 1, 2 NT NT T	J = 0, 1, 2 T T T	J = 0, 1, 2 T T T

Figure 7 actual prediction and hardware prediction

References

1. computer organization and architecture by william stallings, 10th edition
2. https://en.wikipedia.org/wiki/Branch_predictor
3. https://www.cs.umd.edu/~meesh/411/CA-online/chapter/dynamic-branch-prediction/index.html
4. https://www.geeksforgeeks.org/correlating-branch-prediction