Computer Architecture

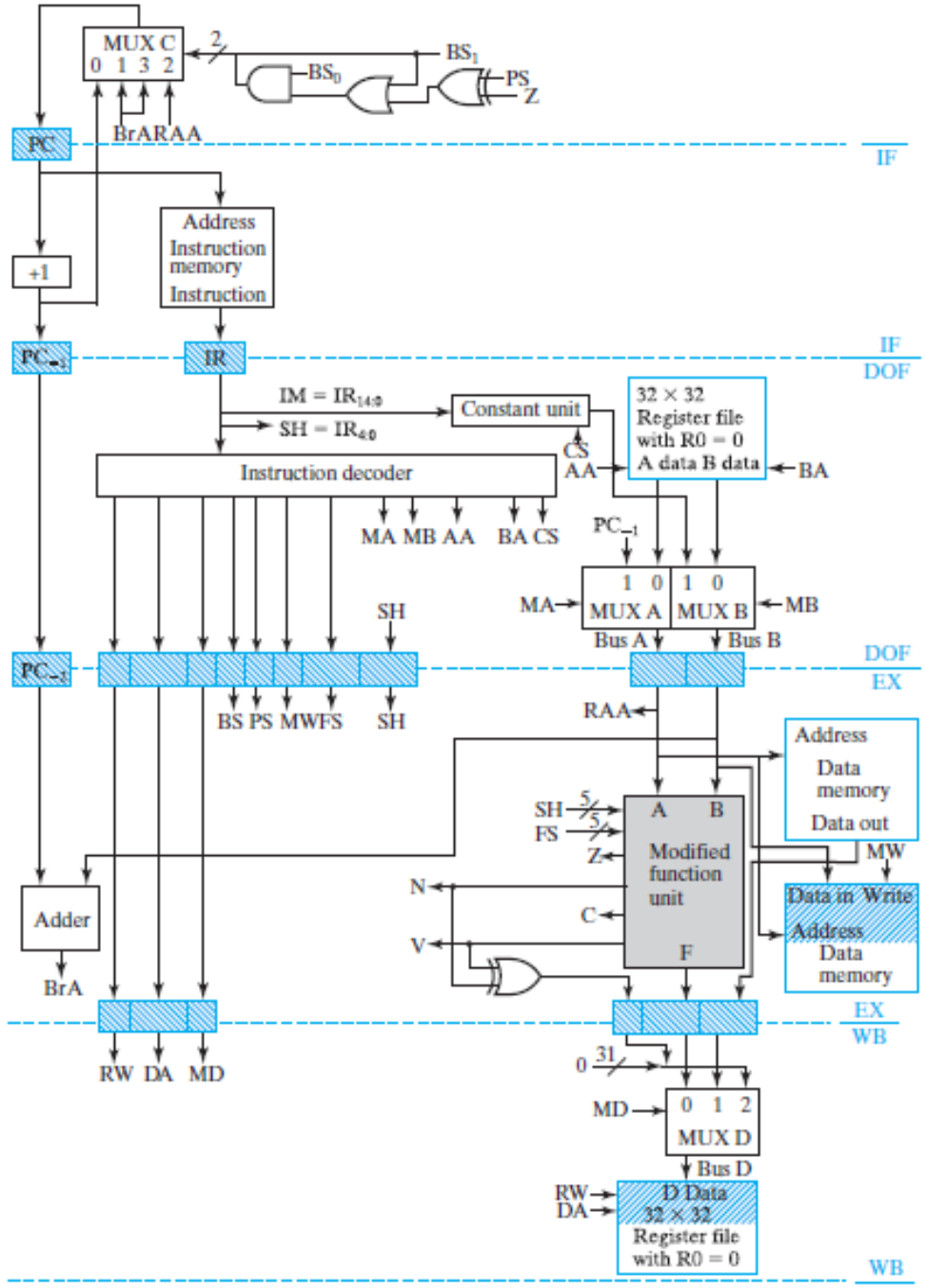
Data and Control Hazards

Dr. Masood Mir

Week 14

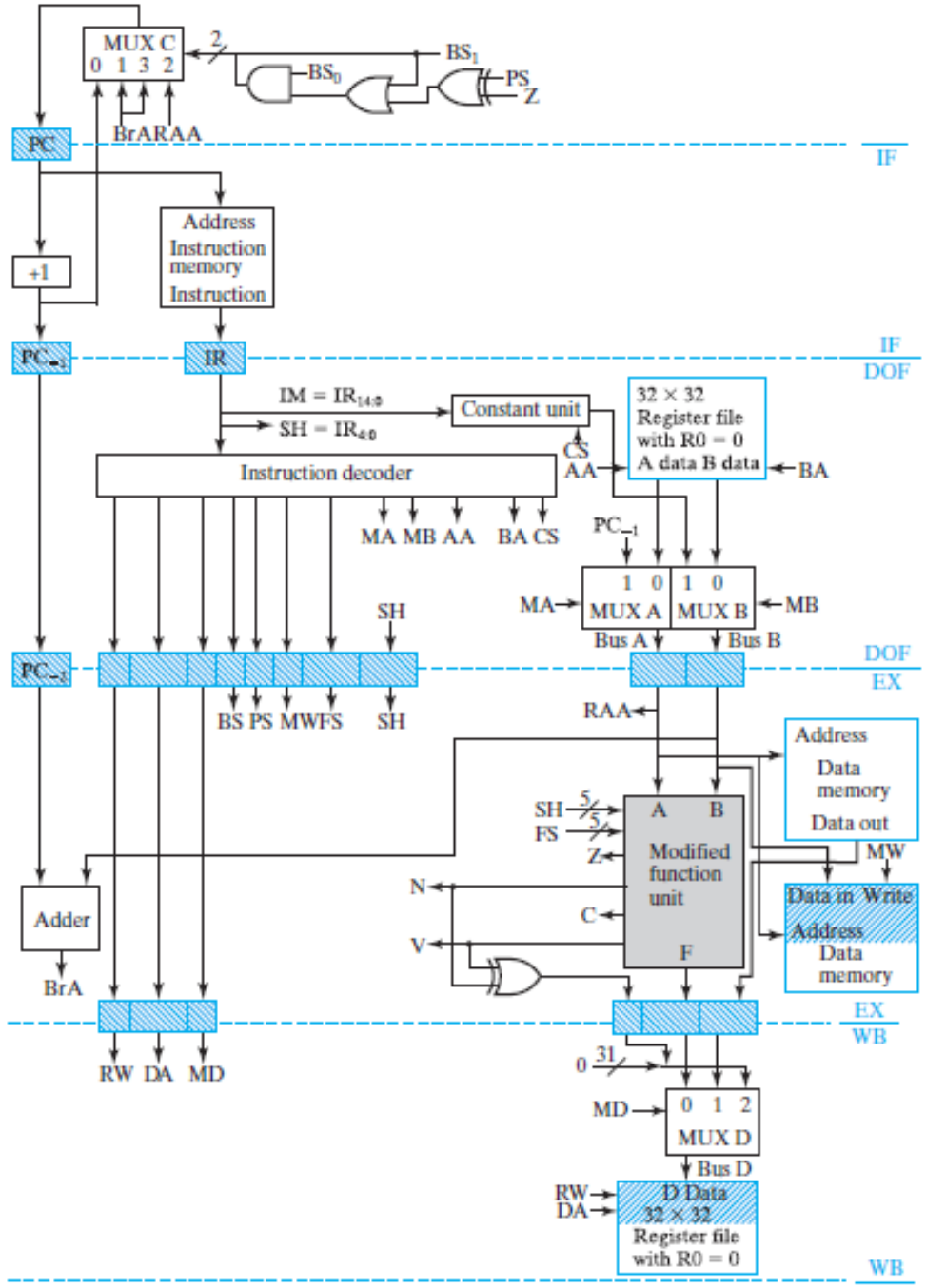
Data Hazards (RISC)

- In the discussion of pipelined CPU, we found that filling and flushing of the pipeline reduced the throughput below the maximum level achievable.
- Unfortunately, there are other problems with pipeline operation that reduce throughput. We will examine two such problems: data hazards and control hazards.
- **Hazards are timing problems** which can cause incorrect and wrong data to be read or written.
- If a subsequent instruction tries to use the result of the operation as an operand before the result is available, it uses the old or stale value, which would give a wrong result.
- To deal with data hazards, either software has to take care of it or the hardware.



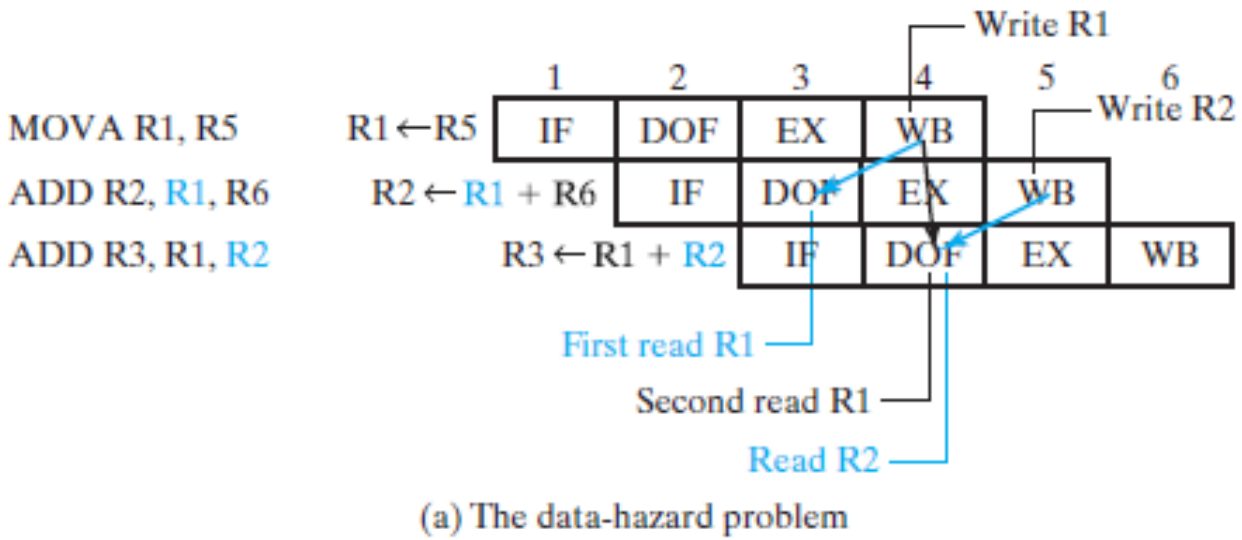
Data Hazards (RISC)

1	MOVA R1, R5
2	ADD R2, R1, R6
3	ADD R3, R1, R2

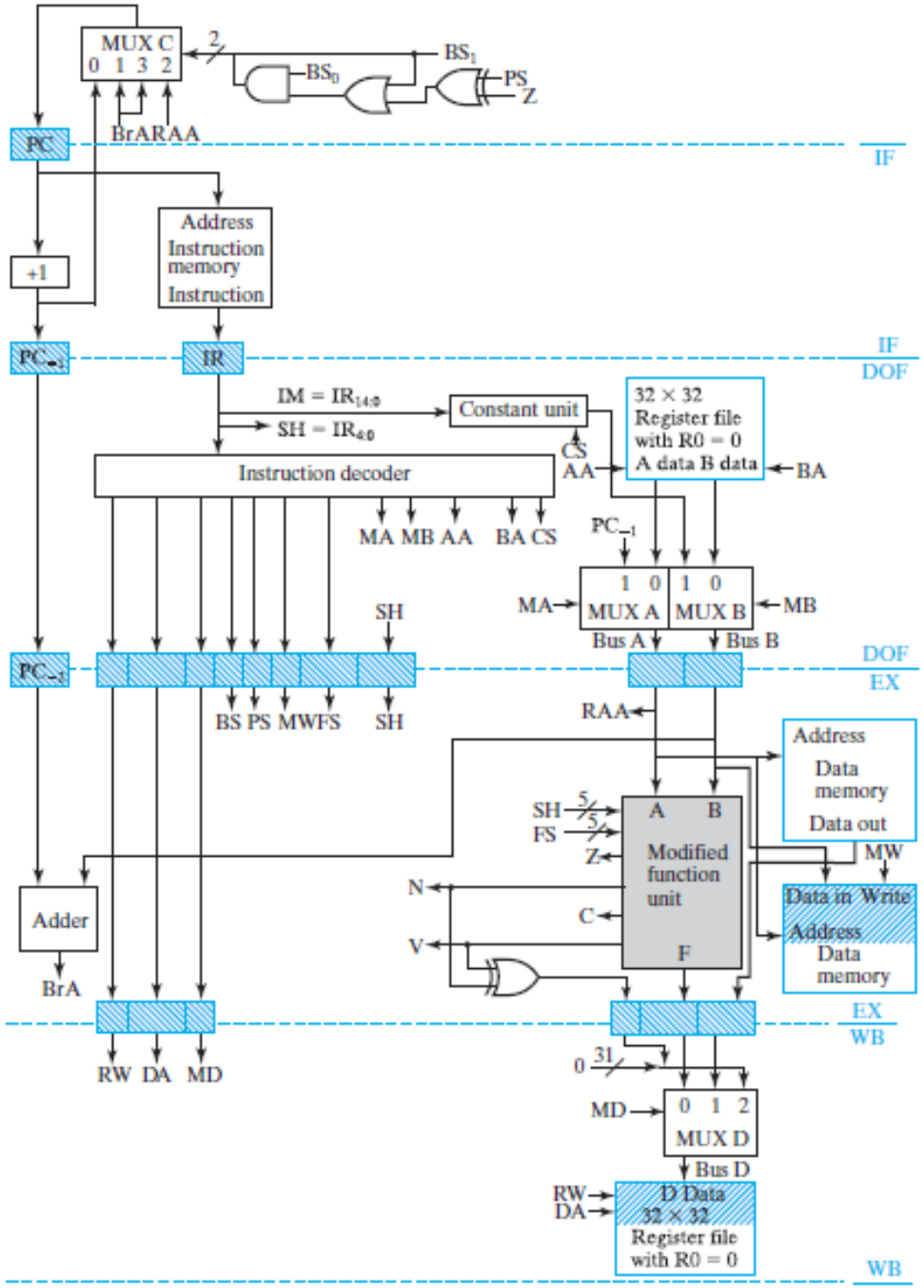


Data Hazards (RISC)

1 MOVA R1, R5
2 ADD R2, R1, R6
3 ADD R3, R1, R2



(a) The data-hazard problem

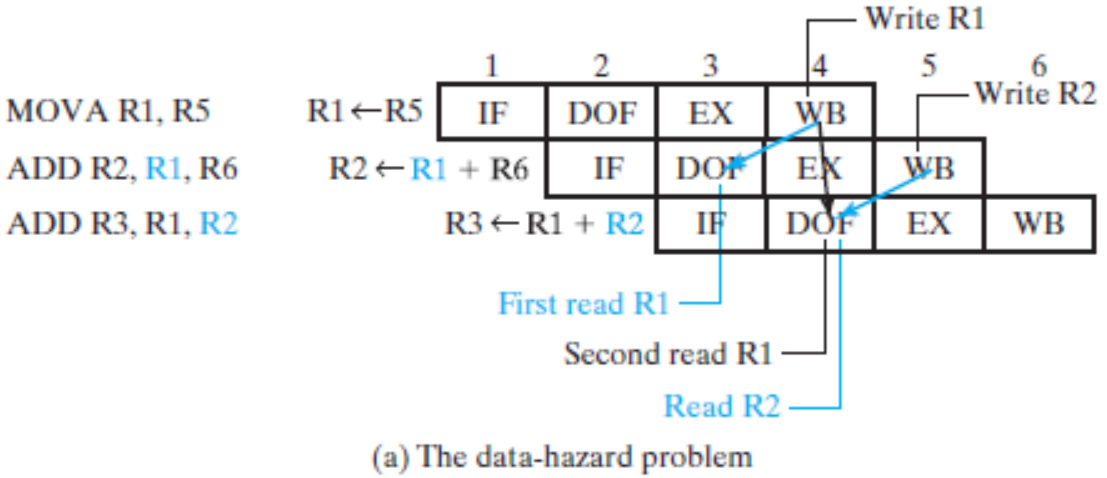


Data Hazards (RISC)

- Two data hazards are illustrated by examining the execution of the following program:

```
1        MOVA R1, R5
2        ADD R2, R1, R6
3        ADD R3, R1, R2
```

- Execution diagram of this program appears in Figure 10-10(a).
- The MOVA instruction places the contents of R5 into **R1** in the **first half of WB in cycle 4**. But, as shown by the blue arrow, the first ADD instruction reads R1 in the **last half of DOF in cycle 3**, one cycle before it is written. Thus, the ADD instruction uses the stale (old) value in R1. The result of this operation is placed in **R2** in the **first half of WB in cycle 5**.
- The second ADD instruction, however, reads both



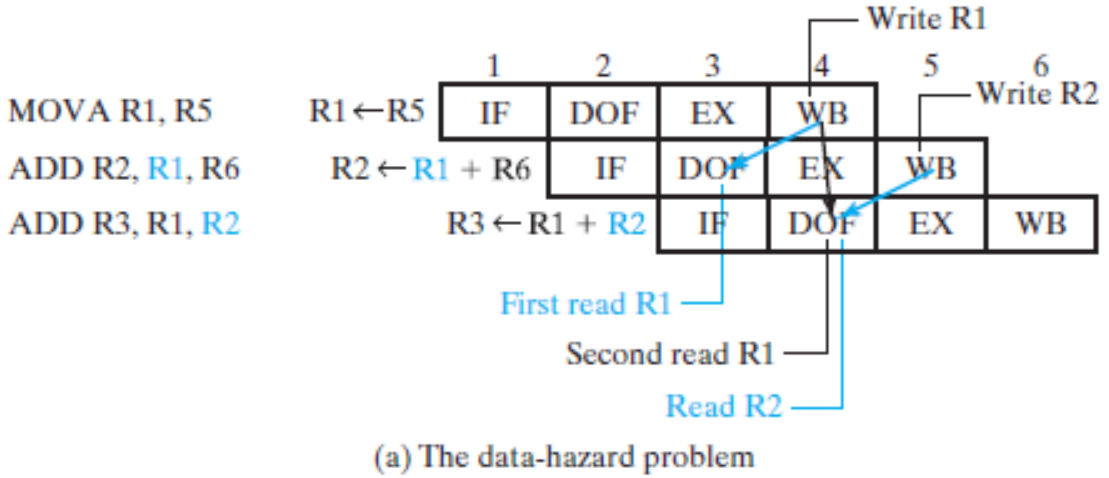
(a) The data-hazard problem

R1 and R2 in the second half of DOF in cycle 4.

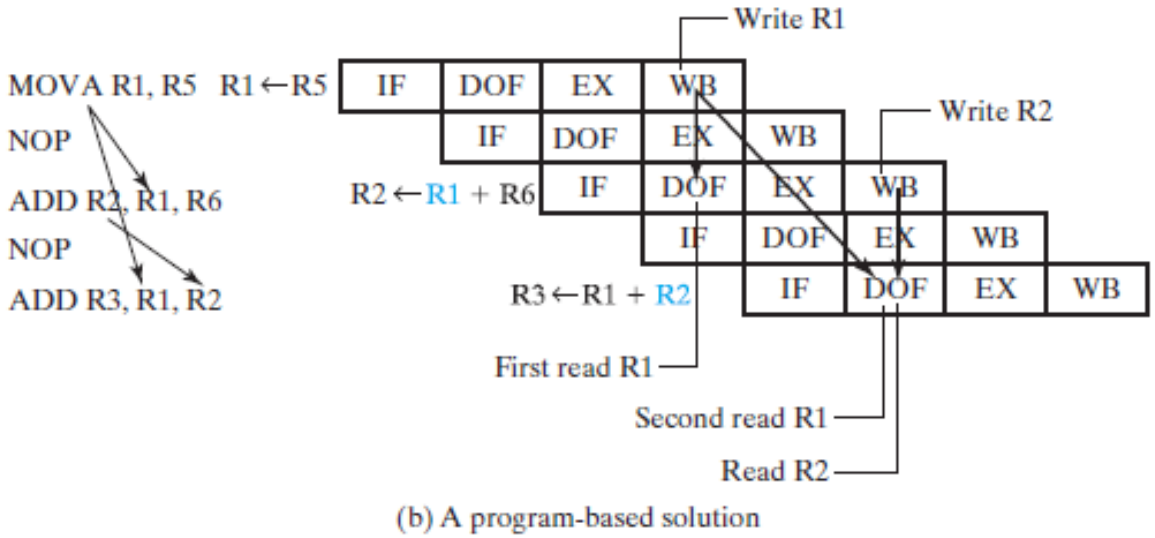
- In the case of R1, the value read was written in the first half of WB in cycle 4 (**Read after write**). So the value read in the second half of cycle 4 is the new value. The **write-back of R2**, however, occurs in the **first half of cycle 5**, after it is read by the next instruction during cycle 4. So R2 has not been updated to the new value at the time it is read.
- This gives **two data hazards**, as indicated by the blue arrows in the figure and highlighted in the program and in the register transfer statements.

Data Hazards (RISC)

- In each of these cases, **the read of the involved register occurs one clock cycle too soon** with respect to the write of that register.
- One possible remedy for data hazards is to have the compiler generate the machine code to **delay instructions** so that new values are available.
- The program is written so that any pending write to a register occurs in the same or an earlier clock cycle than a subsequent read from the register.
- A modification of the simple three-line program that solves the problem is shown in figure b.
- **No-operation (NOP) instructions are inserted, when data hazard is detected**, to delay the respective reads relative to the writes by one clock cycle.



(a) The data-hazard problem



(b) A program-based solution

Data Forwarding

- Another hardware solution of Data Hazards is to do "**Data Forwarding**".
- Data Forwarding is based on the answer to the following question: When a data hazard is detected, **is the result available somewhere else in the pipeline**, so that it can be used immediately in the operation having the data hazard?
- The answer is "almost." The result will be on Bus D, but it is not available until the next clock cycle. The result is to be written into the destination register during that clock cycle. The information needed to form the result, however, is available on the inputs to the pipeline platform that provides the inputs to MUX D.
- So, instead of reading the operand from the register file, we use data forwarding to read the operand directly one clock cycle earlier.

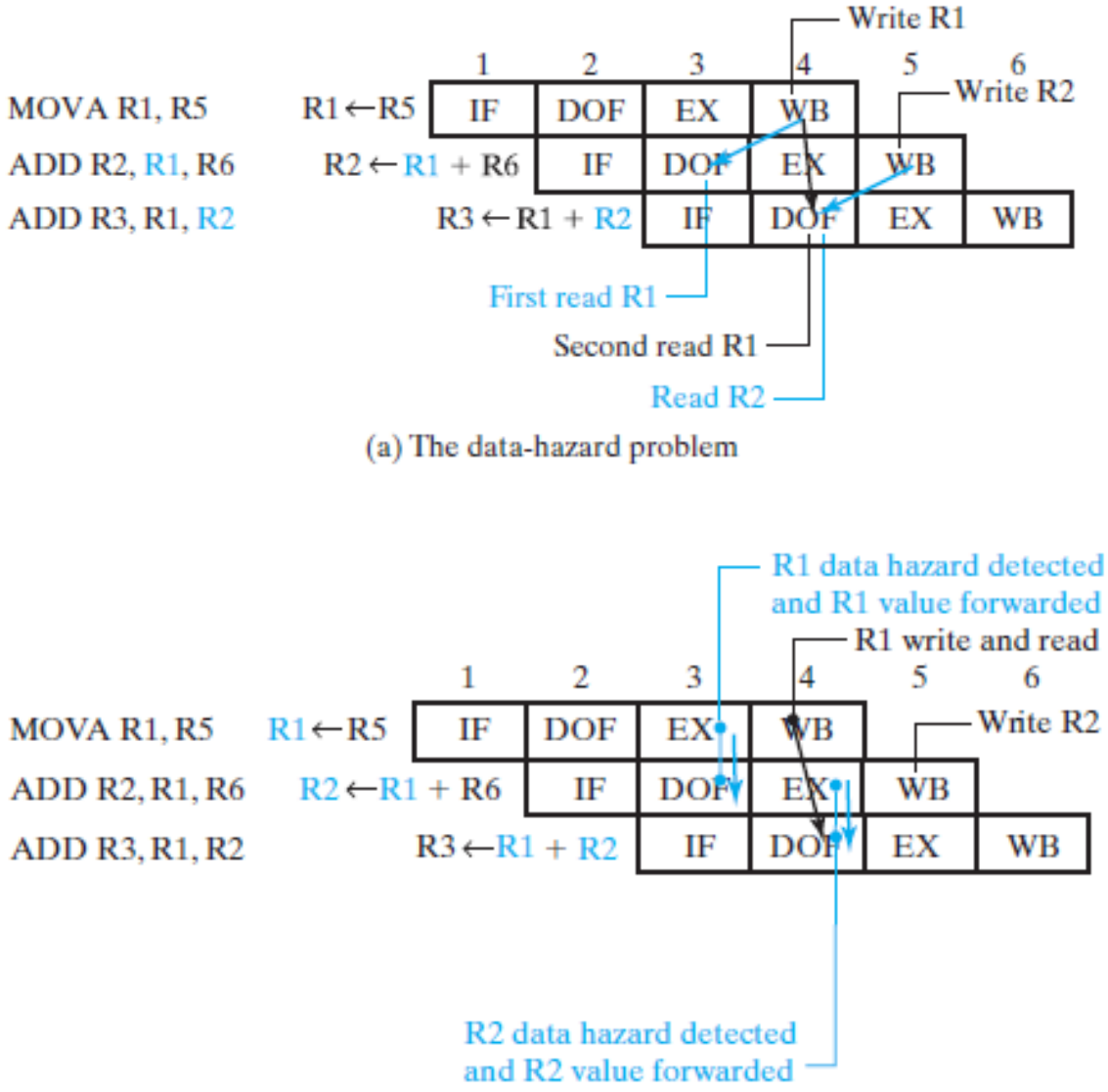


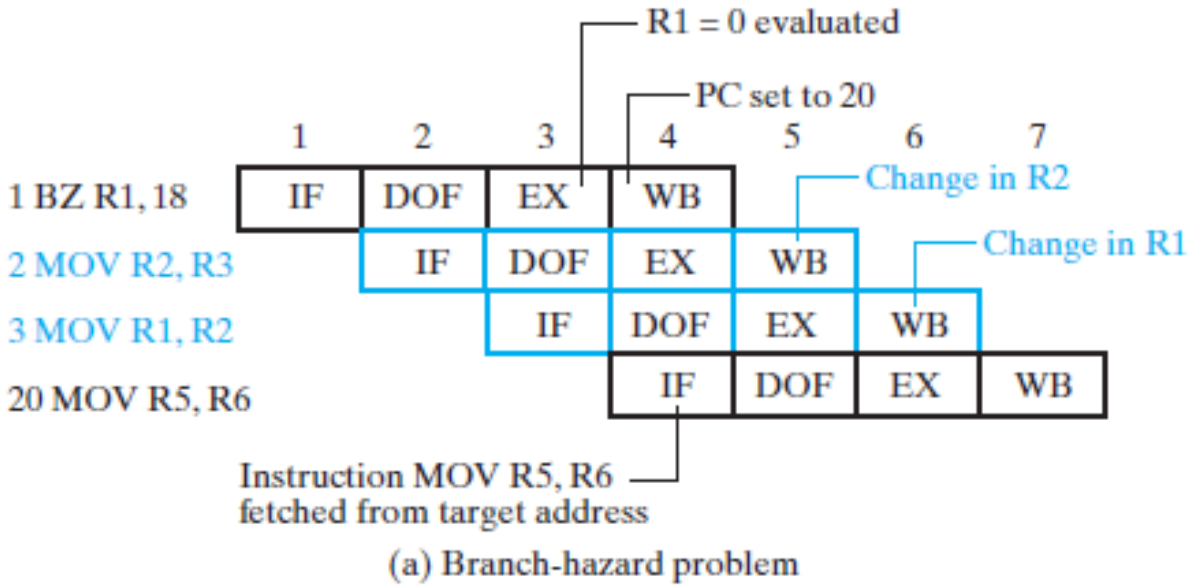
FIGURE 10-14
Example of Data Forwarding

Control Hazards

- Control hazards are associated with branches in the **control flow of the program**.
- The following program containing a conditional branch illustrates a control hazard:

```
1     BZ R1, 18
2     MOVA R2, R3
3     MOVA R1, R2
4     MOVA R4, R2
20    MOVA R5, R6
```

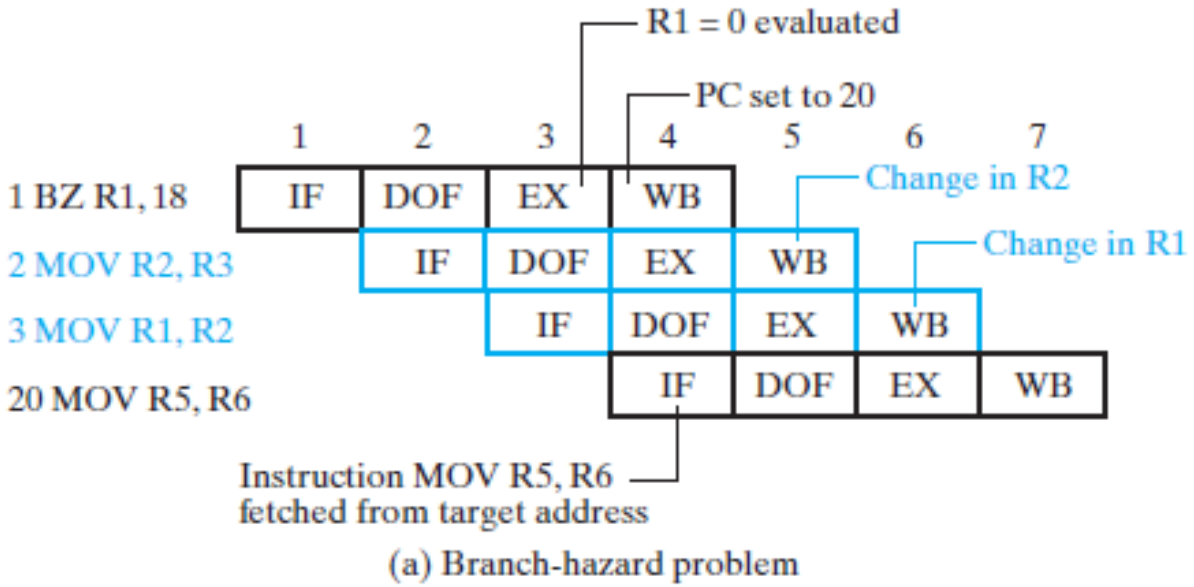
- If R1 is zero, then a branch to the instruction in location 20 (recall that addressing is PC relative) is to occur, skipping the instructions in locations 2 and 3.
- If R1 is nonzero, then the instructions in stages 2 and 3 are to be executed in sequence.



(a) Branch-hazard problem

Control Hazards

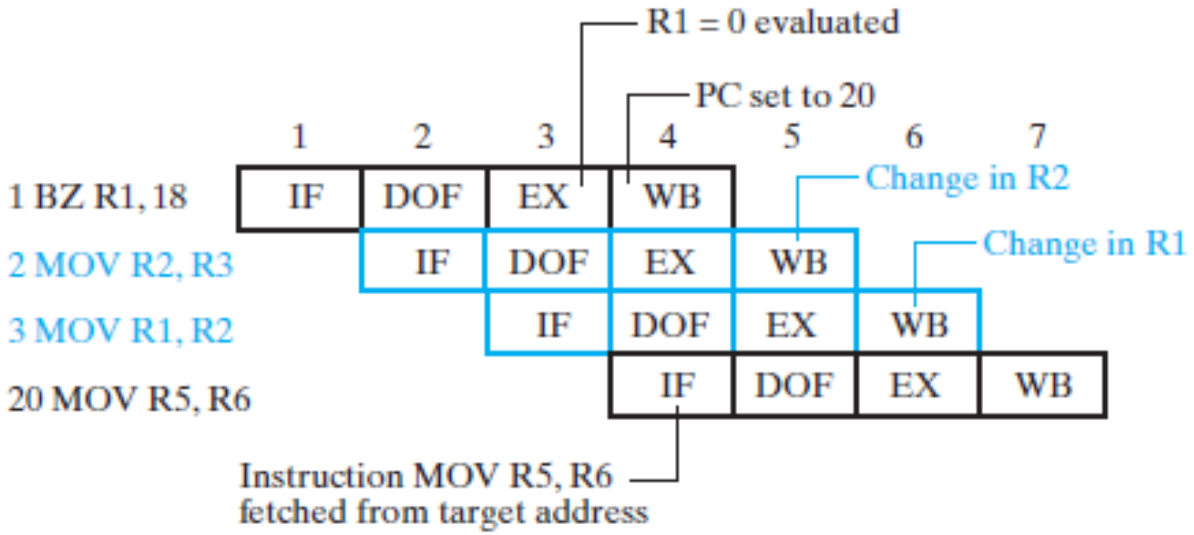
- **Assume** that the branch is taken to address 20 because **R1 is equal to zero**.
- The fact that R1 equals 0 is not detected until **EX in cycle 3** of the first instruction. So the PC is set to 20 on the clock edge at the end of cycle 3. But the MOVA instructions in stages 2 and 3 are into the EX and DOF stages.
- Thus, corrective action is required Otherwise, these instructions will complete execution and produce incorrect results. This situation is one form of a **control hazard**.
- One solution is to **insert NOP instructions** to deal with control hazards just as they were with data hazards. The insertion of NOPs has to be performed by the compiler while generating the machine-language program.



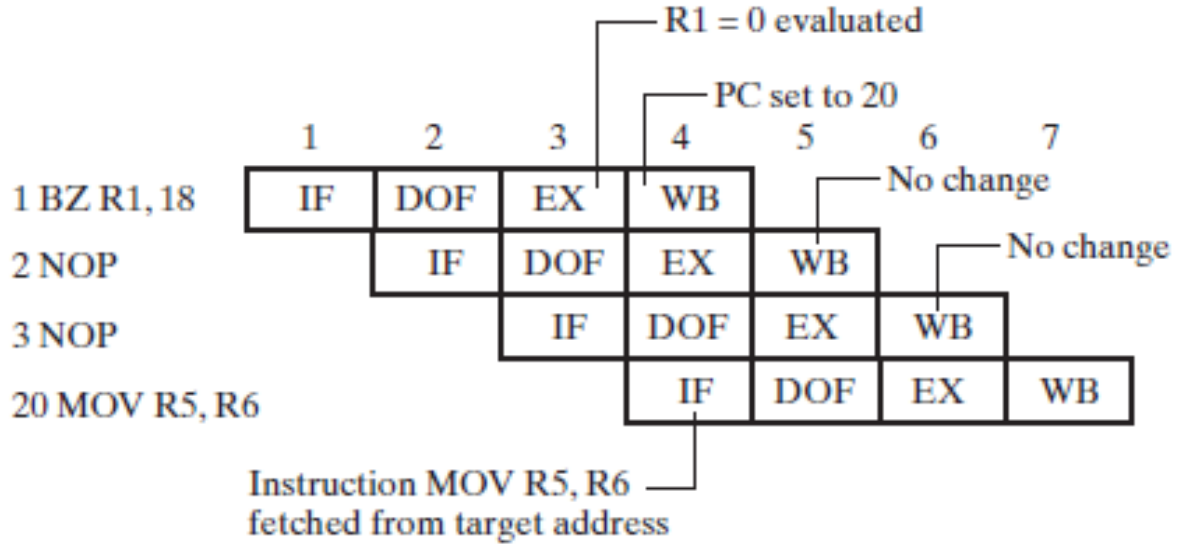
(a) Branch-hazard problem

Control Hazards

- A solution of the simple three-line program to avoid control hazard is given in figure b.
- **Two NOPs are inserted after the branch instruction BZ.** These two NOPs can be performed regardless of whether the branch is taken in the EX stage of BZ in cycle 3, with no adverse effects on the correctness of the program.
- When control hazards in the CPU are handled in this manner by programming, the branch hazard dealt with by the NOPs is referred to as a "**Delayed Branch**", since branch execution is delayed by two clock cycles in this CPU.
- The NOP solution increases the time required to process the simple program by two clock cycles, regardless of whether the branch is taken.
- These wasted cycles can sometimes be avoided by **rearranging the order** of instructions.



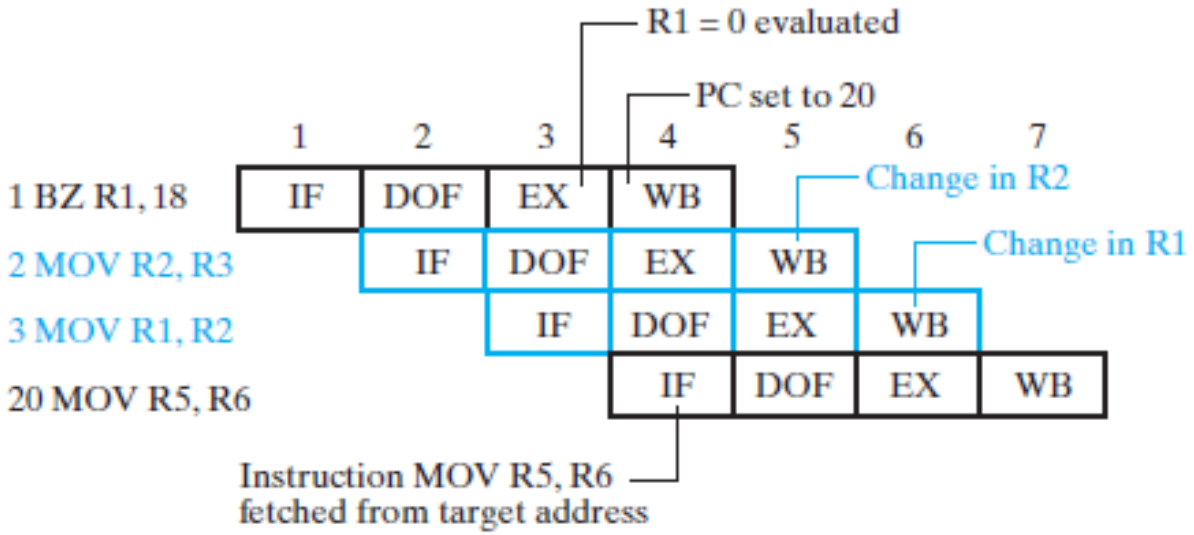
(a) Branch-hazard problem



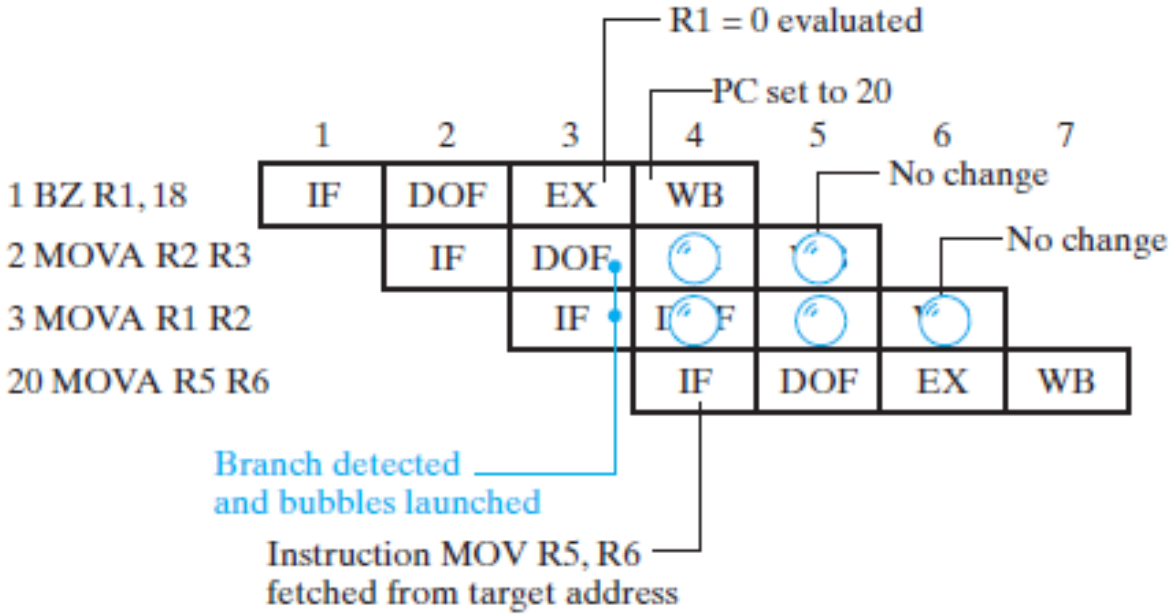
(b) Program-based solution

Control Hazards

- A second hardware solution is to use **branch prediction**.
- This method predicts that **branches will never be taken**. Thus, instructions will be fetched and decoded and operands fetched on the basis of the addition of 1 to the value of the PC.
- These actions occur until it is known during the execution cycle whether the branch in question will be taken.
- If the branch is not taken, the instructions already in the pipeline due to the prediction will be allowed to proceed.
- If the branch is taken, the instructions following the branch instruction **need to be canceled**. Usually, the cancellation is done by inserting bubbles into the execution and write-back stages for these instructions.

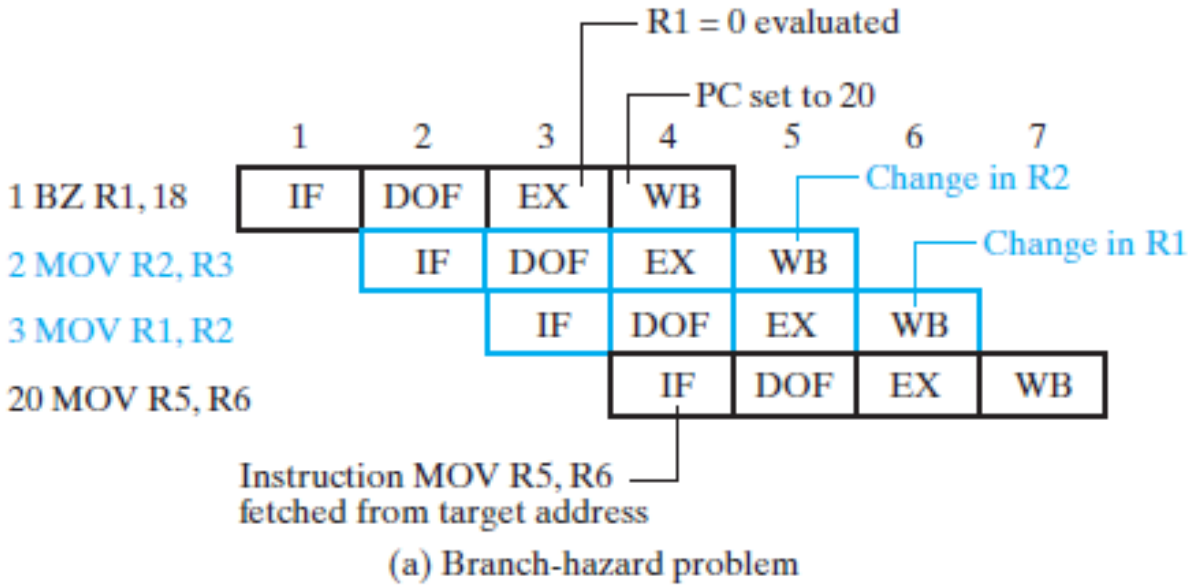


(a) Branch-hazard problem



Control Hazards

- On the basis of the prediction that the branch will not be taken, the two MOVA instructions after BZ are fetched. These actions take place in cycles 2 and 3.
- In **cycle 3**, the condition upon which the branch is based has been evaluated, and it is found that R1 = 0. Thus, the branch is to be taken.
- At the end of cycle 3, the **PC is set to 20**, and the instruction fetch in cycle 4 is performed using the new value of the PC.
- In cycle 3, the fact that the branch is taken has been detected, and **bubbles are inserted into the pipeline for instructions 2 and 3**. Proceeding through the pipeline, these bubbles have the same effect as two NOP instructions. Because the NOPs are not present in the program, there is no delay when the branch is not taken.



(a) Branch-hazard problem