Pipeline Hazards

Pipeline Hazards

- There are situations, called hazards, that prevent the next instruction in the instruction stream from executing during its designated cycle
- There are three classes of hazards
  - Structural hazard
  - Data hazard
  - Branch hazard

Structural Hazards. They arise from resource conflicts when the **hardware cannot support** all possible combinations of instructions in simultaneous overlapped execution.

Data Hazards. They arise when an instruction depends on the **result of a previous instruction** in a way that is exposed by the overlapping of instructions in the pipeline.

Control Hazards.They arise from the **pipelining of branches and other instructions** that change the PC.

What Makes Pipelining Hard?

- **Power failing,**
- **Arithmetic overflow,**
- **I/O device request,**
- **OS call,**
- **Page fault**

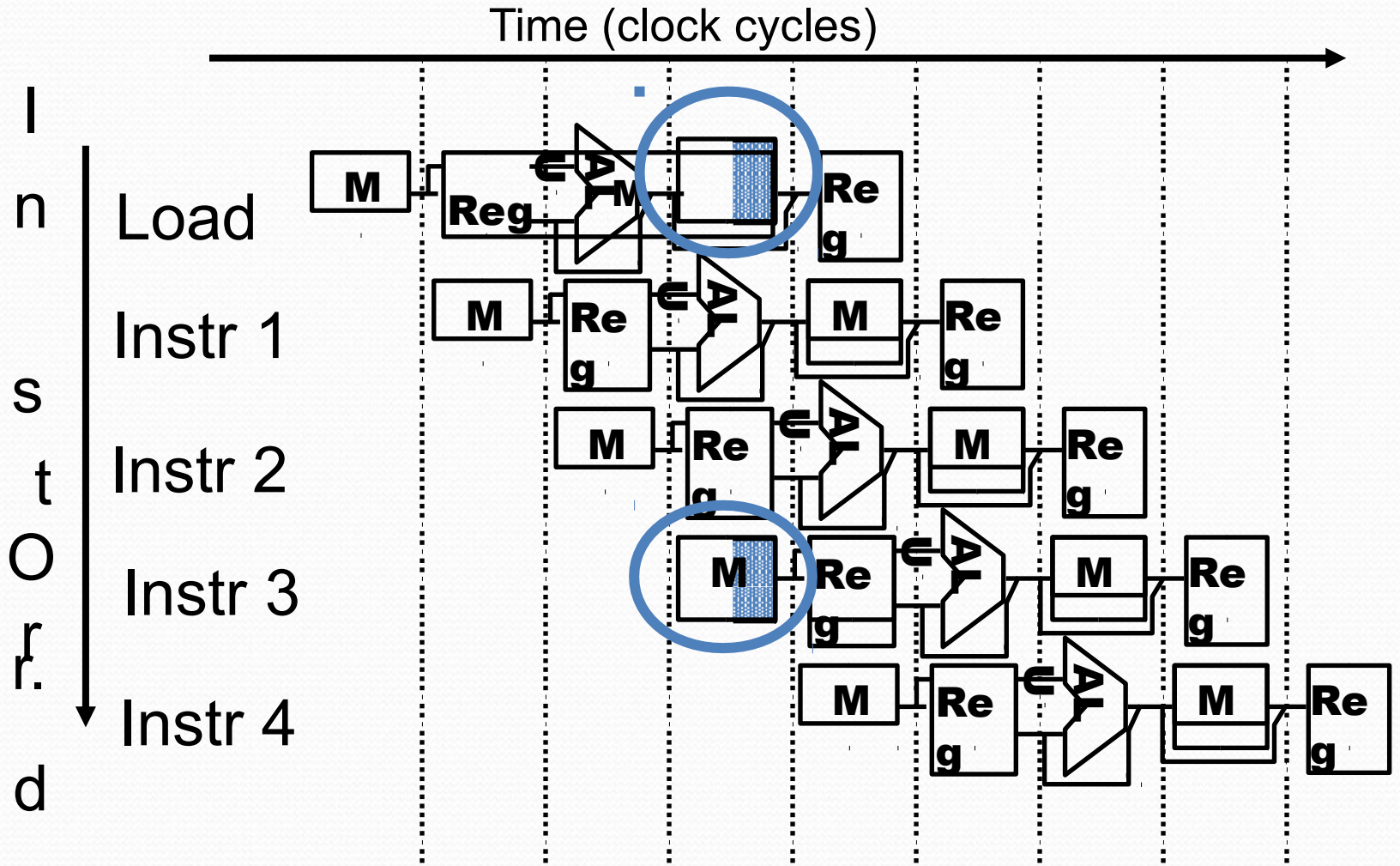
Pipeline Hazards

- **Structural hazard**
  - Resource conflicts when the hardware cannot support all possible combination of instructions simultaneously
- **Data hazard**
  - An instruction depends on the results of a previous instruction
- **Branch hazard**
  - Instructions that change the PC

Structural hazard

- Some pipeline processors have shared a single-memory pipeline for data and instructions

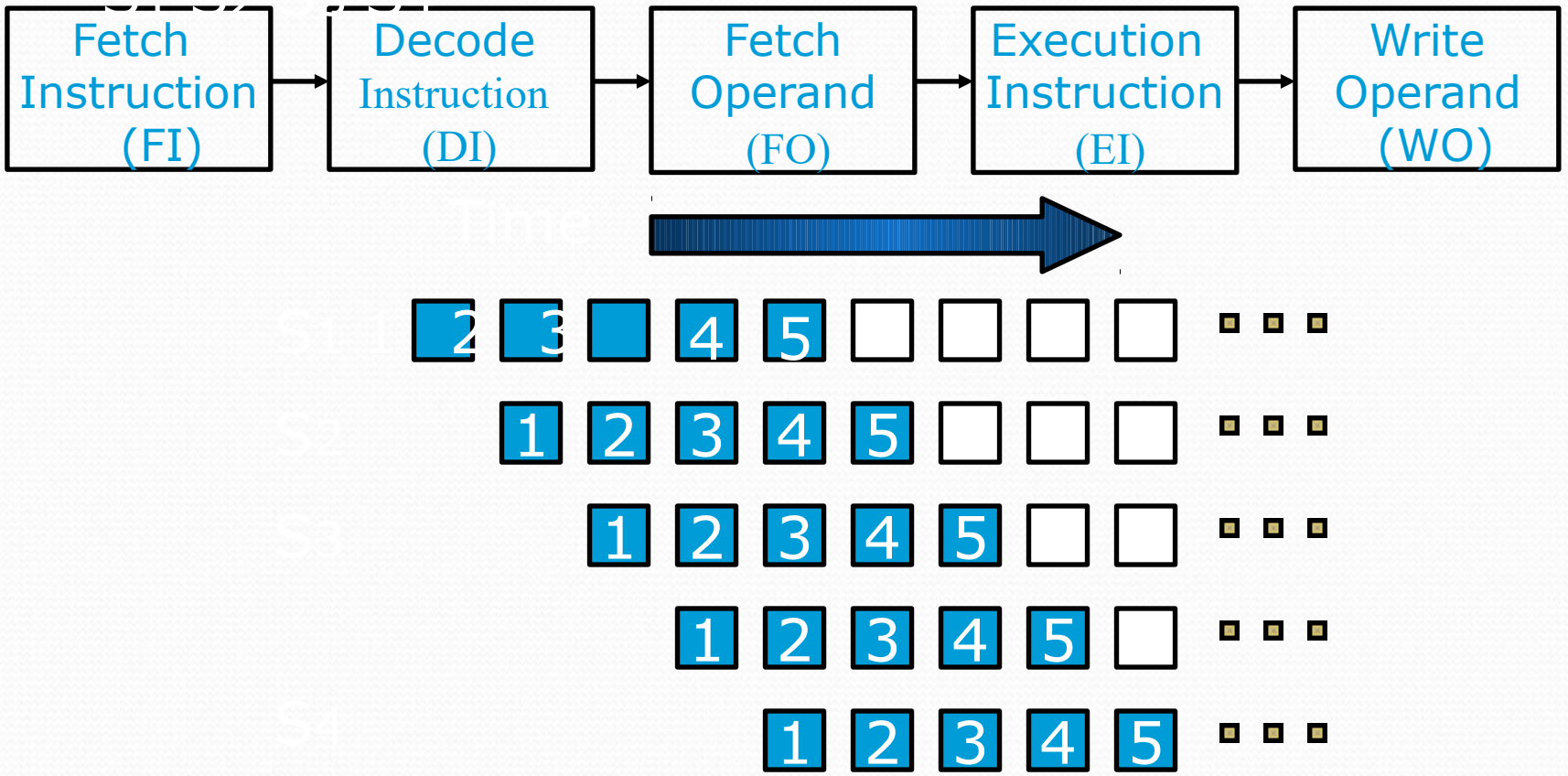
Single Memory is a Structural Hazard



- Can't read same memory twice in same clock cycle

Structural hazard

Memory data fetch requires on FI and FO



Structural hazard

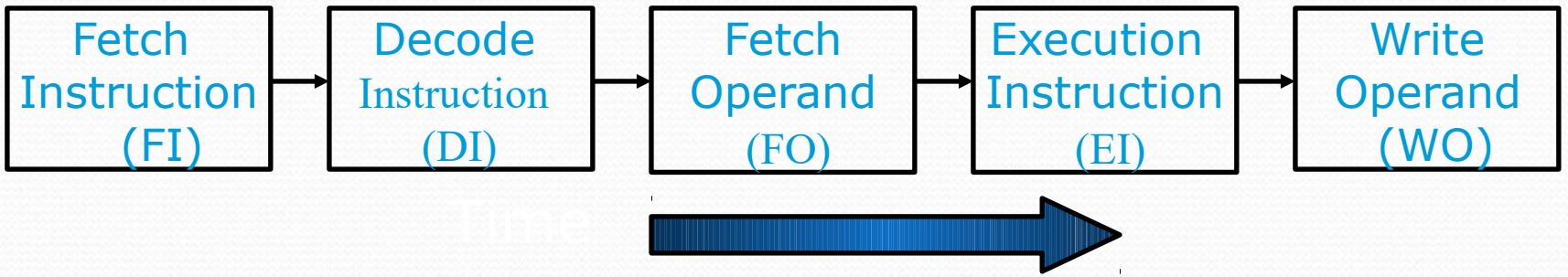
- To solve this hazard, we “stall” the pipeline until the resource is feed
- A stall is commonly called pipeline bubble, since it floats through the pipeline taking space but carry no useful work

Structural Hazards limit performance

- Example: if 1.3 memory accesses per instruction (30% of instructions execute loads and stores) and only one memory access per cycle then
  - Average CPI ≥ 1.3
  - Otherwise datapath resource is more than 100% utilized

Structural Hazard Solution: Add more Hardware

Structural hazard



Data hazard

Example:

ADD R1 ← R2+R3

SUB R4 R1-R5

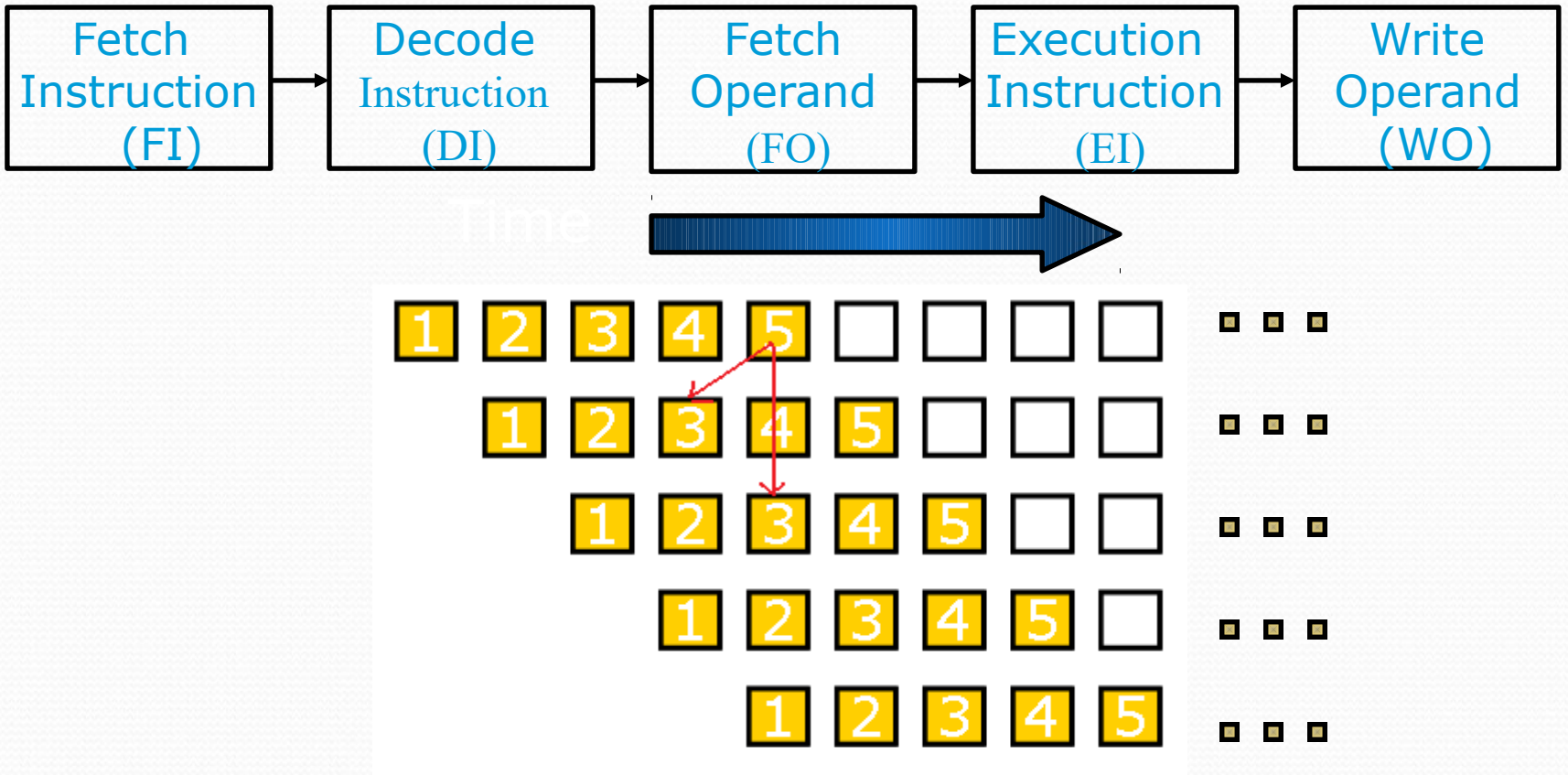
AND R6 R1 AND R7

OR R8 R1 OR R9

XOR R10 R1 XOR R11

Data hazard

FO: fetch data value WO: store the executed value

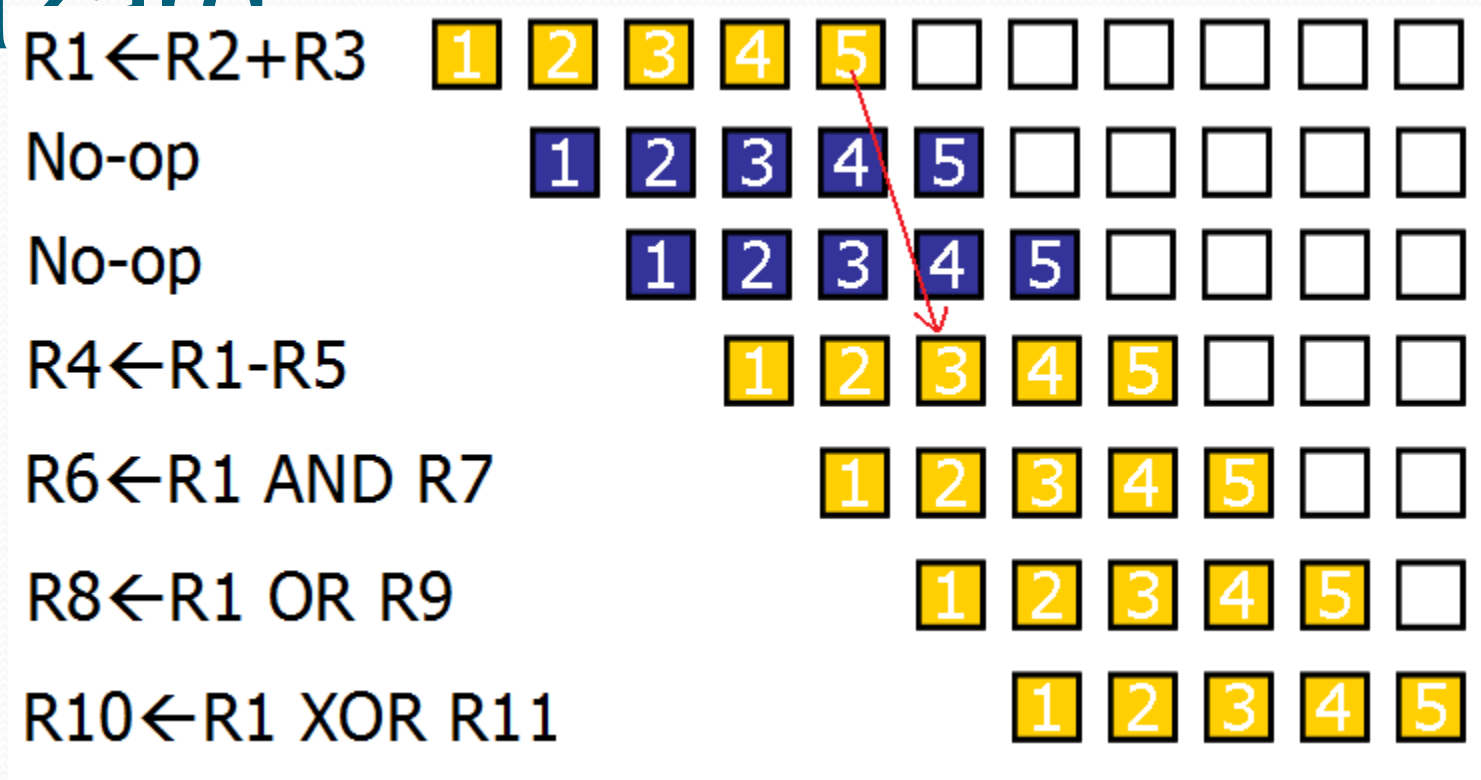


Data hazard

- Delay load approach inserts a no-operation instruction to avoid the data conflict

ADD	R1←
No-op	R2+R3
No-op	R4← R1-R5
SUB	R6← R1 AND R7
AND	R8← R1 OR R9
OR	R10← R1 XOR
XOR	R11

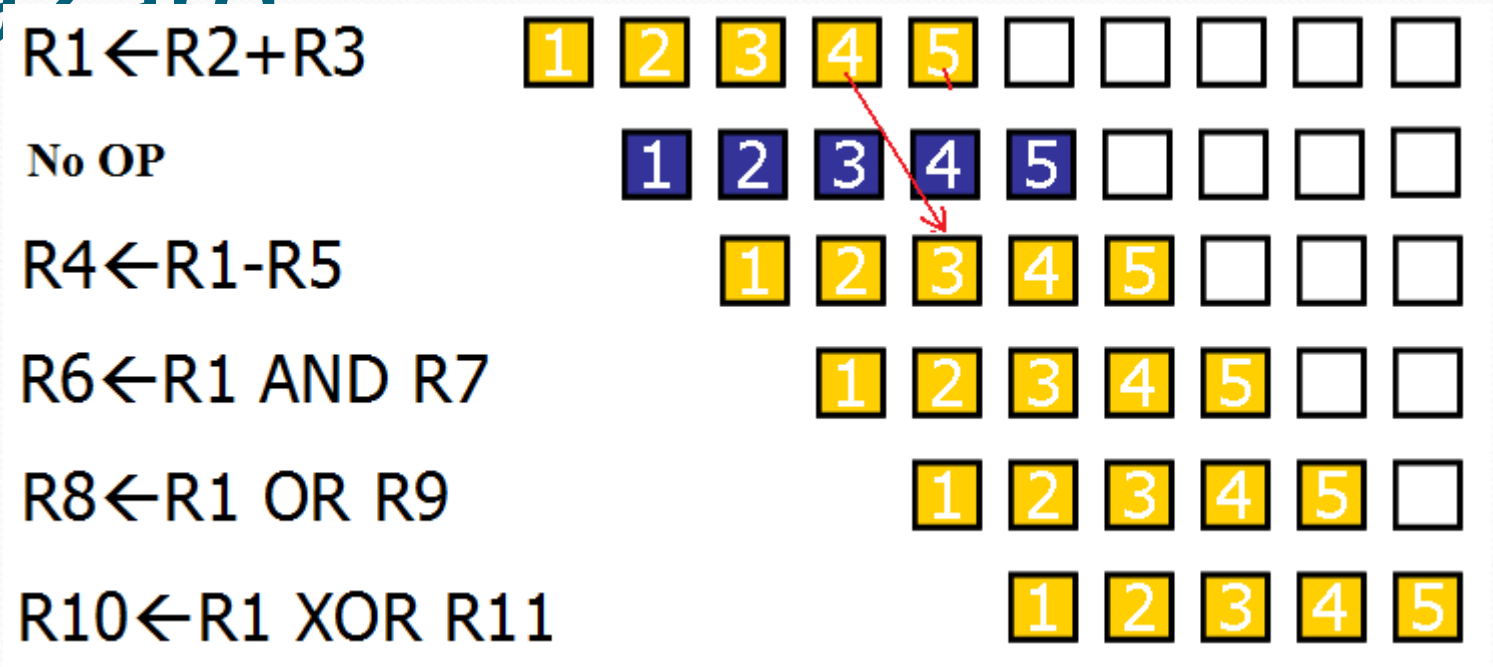
Data hazard


R1←R2+R3
No-op
No-op
R4←R1-R5
R6←R1 AND R7
R8←R1 OR R9
R10←R1 XOR R11
1 2 3 4 5

Data hazard

- It can be further solved by a simple hardware technique called ***forwarding* (also called *bypassing* or *short-circuiting*)**
- The insight in forwarding is that the result is not really needed by SUB until the ADD execute completely
- If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the results in ALU instead of from memory

Data hazard



Data Hazard Classification

- Three types of data hazards
  - RAW : Read After Write
  - WAW : Write After Write
  - WAR : Write After Read

- RAR : Read After Read
  - Is this a hazard?

Read After Write (RAW)

- A read after write (RAW) data hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved.
- This can occur because even though **an instruction is executed after a previous instruction**, the previous instruction has not been completely processed through the pipeline.

example:			
i1.	R2	<-	R1 + R3
i2.	R4	<-	R2 + R3

Write After Read (WAR)

- A write after read (WAR) data hazard represents a problem with concurrent execution.

For example:

```
i1.     R4 <- R1 +
i2.    R5  R5 <- R1
       + R2
```

Write After Write (WAW)

- A write after write (WAW) data hazard may occur in a concurrent execution environment.

example:

```
i1.    R2 <- R4 +
R7   i2.   R2 <- R1
+ R3
```

We must delay the WB (Write Back) of `i2` until the execution of `i1`

Branch hazards

- Branch hazards can cause a greater performance loss for pipelines

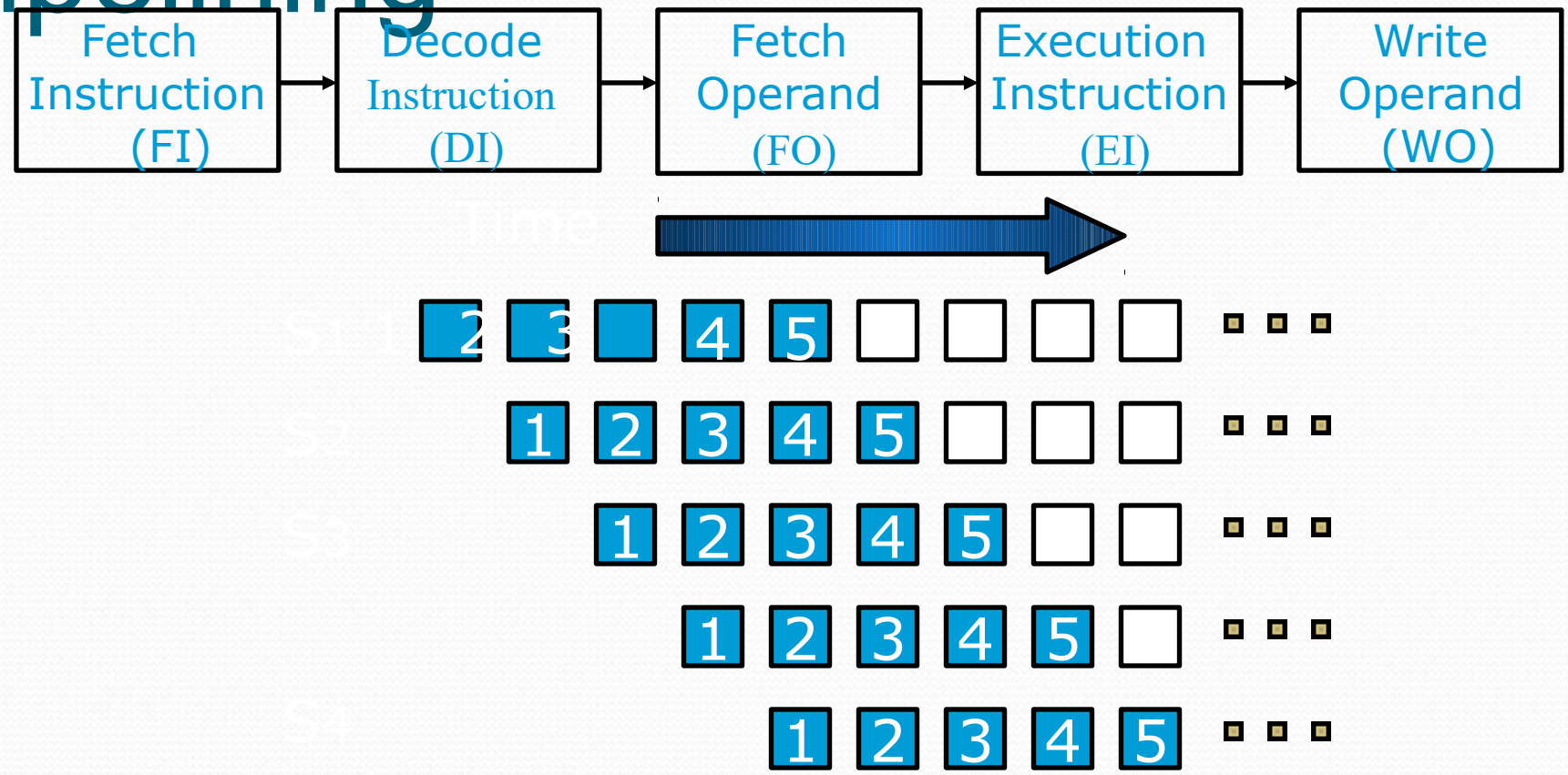
- When a branch instruction is executed, it may or may not change the PC

- **If a branch changes the PC to its target address, it is a *taken* branch Otherwise, it is *untaken***

Branch hazards

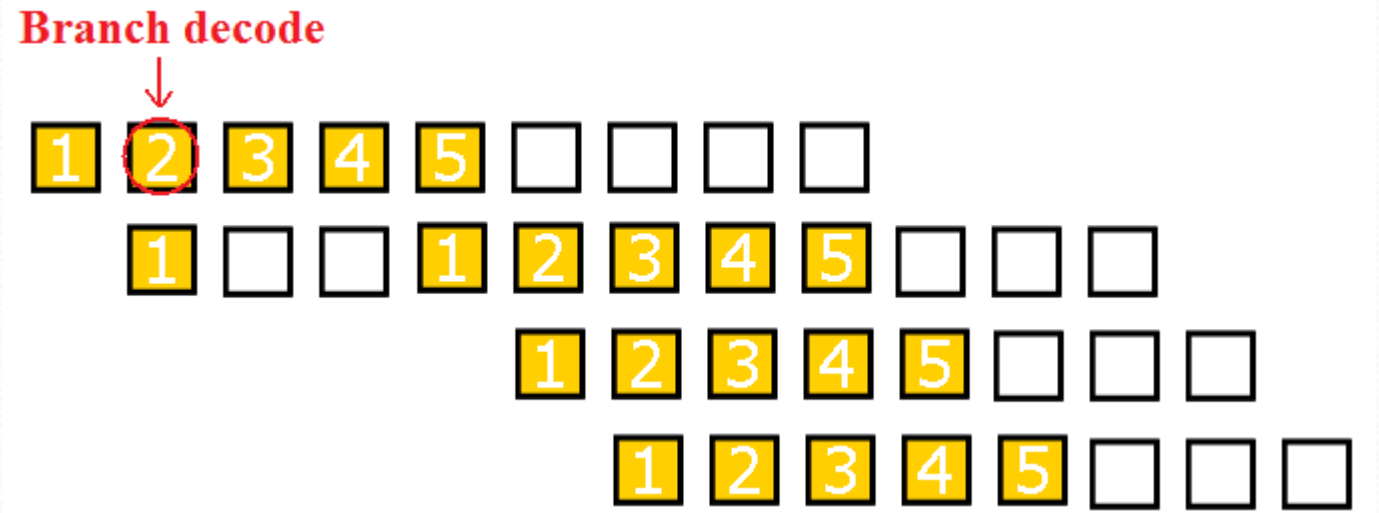
- There are **FOUR** schemes to handle branch hazards
  - **Freeze scheme**
  - **Predict-untaken scheme**
  - **Predict-taken scheme**
  - **Delayed branch**

5-Stage Pipelining



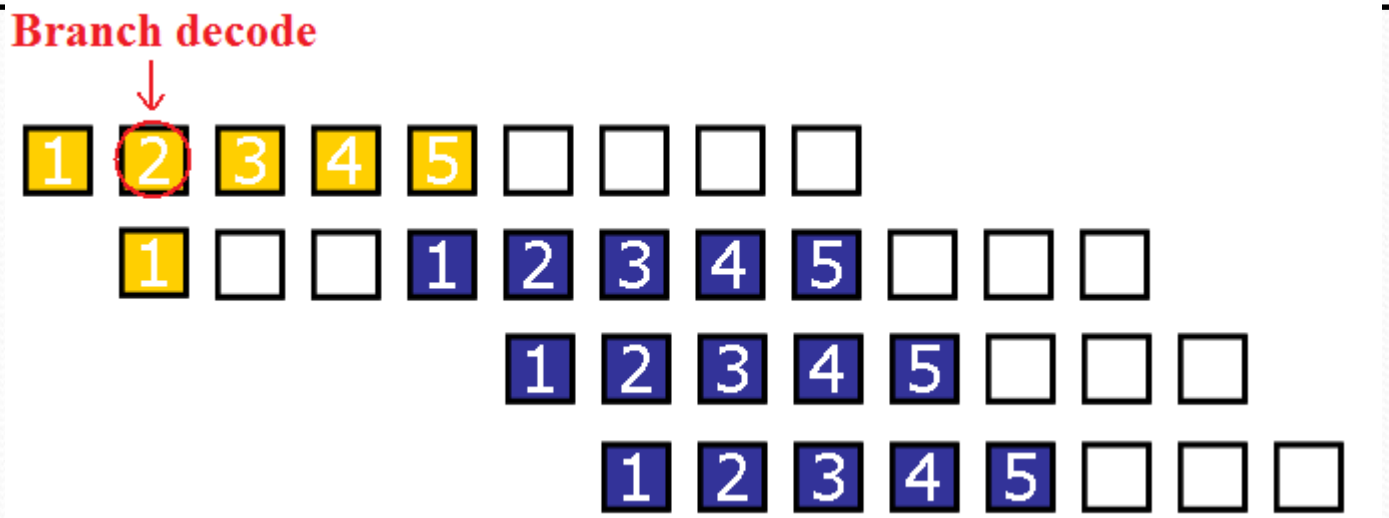
Branch (Untaken approach)

- **The simplest method of dealing with branches is to redo the fetch following a branch**



Branch (Take care approach)

- The simplest method of dealing with branches is to redo the fetch following a branch



Branch (Take/Freeze approach)

- **The simplest scheme to handle branches is to *freeze* the pipeline holding or deleting any instructions after the branch until the branch destination is known**

- The attractiveness of this solution lies primarily in its simplicity both for hardware and software

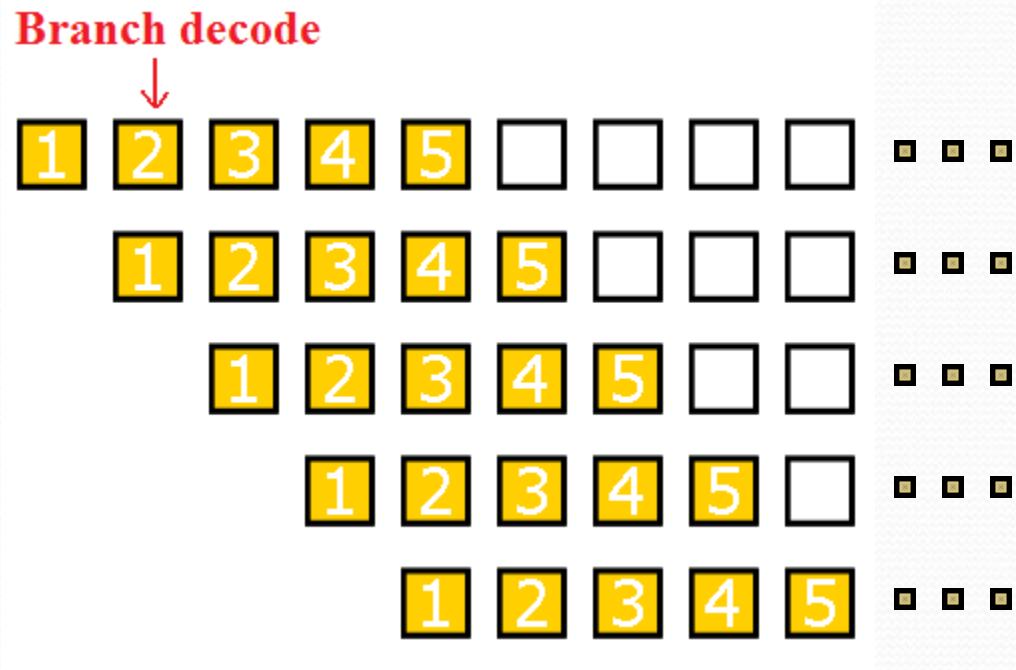
Branch (Predicted-untaken)

- **A higher performance, and only slightly more complex, scheme is to treat every branch as not taken**

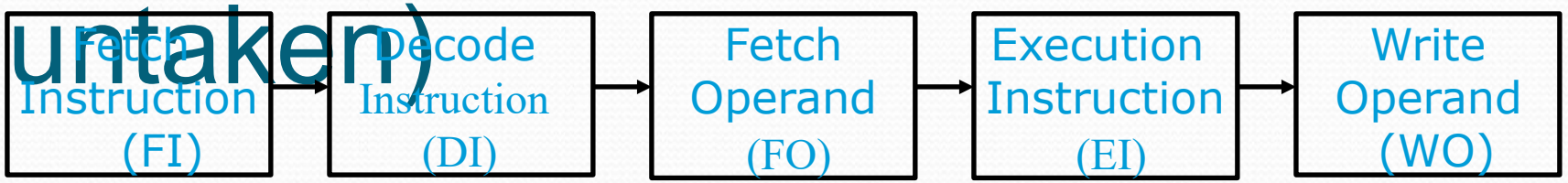
- It is implemented by continuing to fetch instructions as if the branch were normal instruction

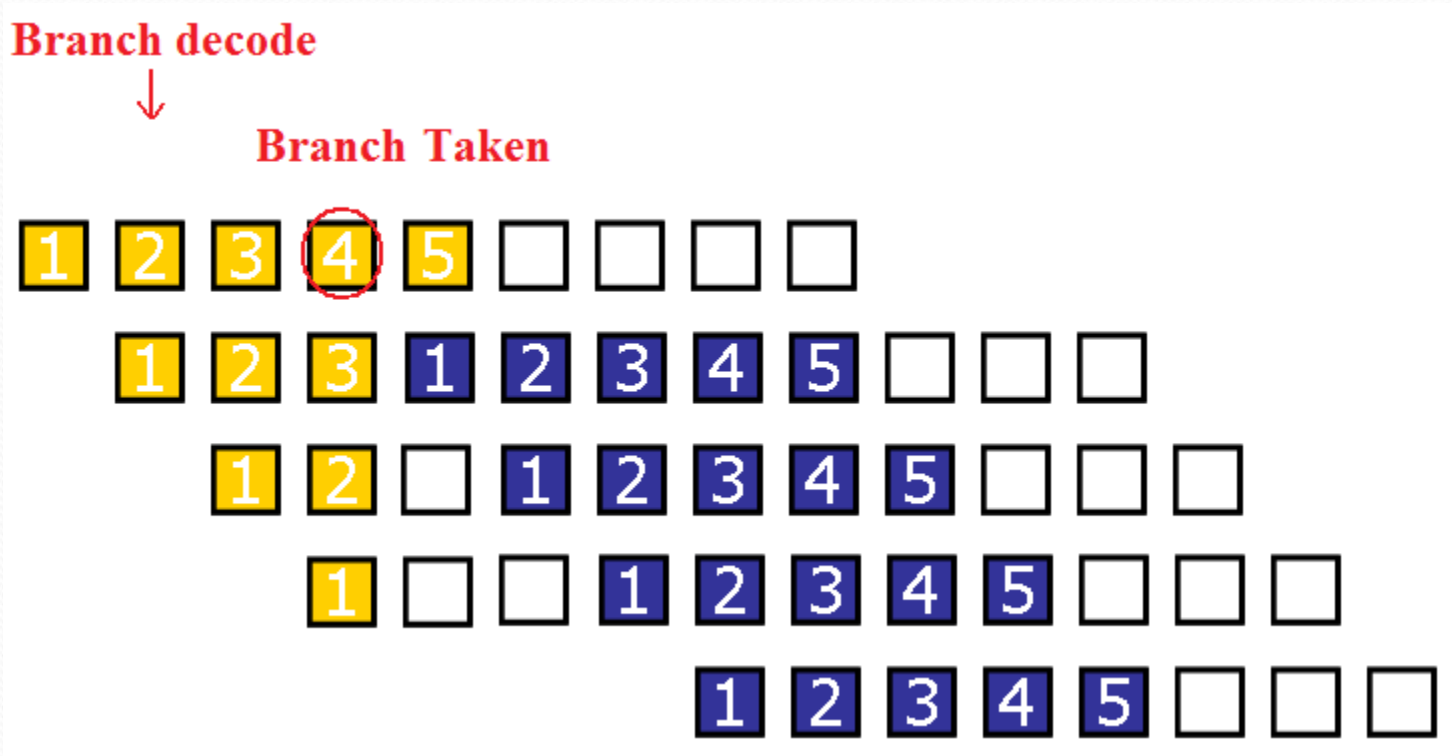
- **The pipeline looks the same if the branch is not taken**
- **If the branch is taken, we need to redo the fetch instruction**

Branch (Predicted-untaken)



Branch (Predicted-taken/ untaken)



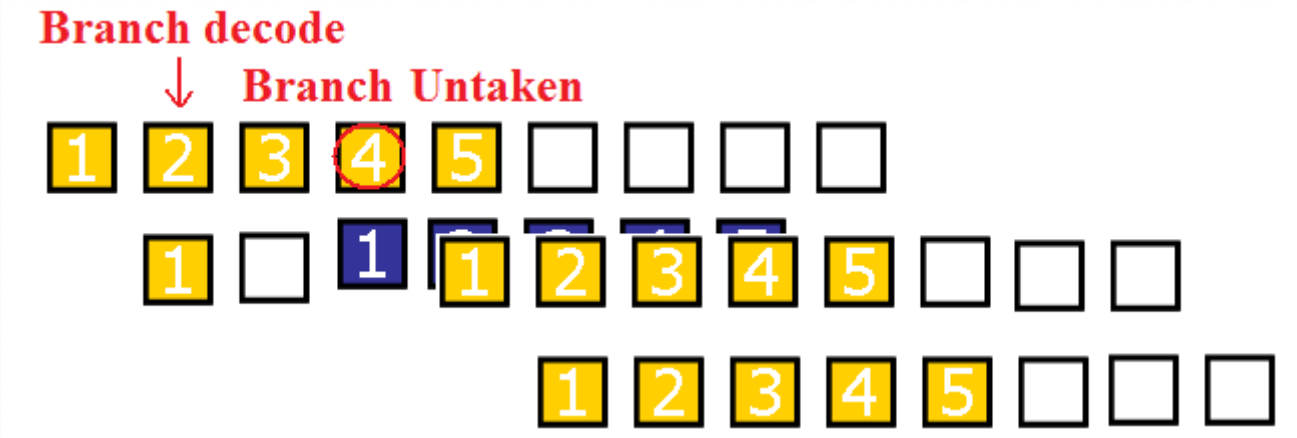


Branch (Predicted-taken)

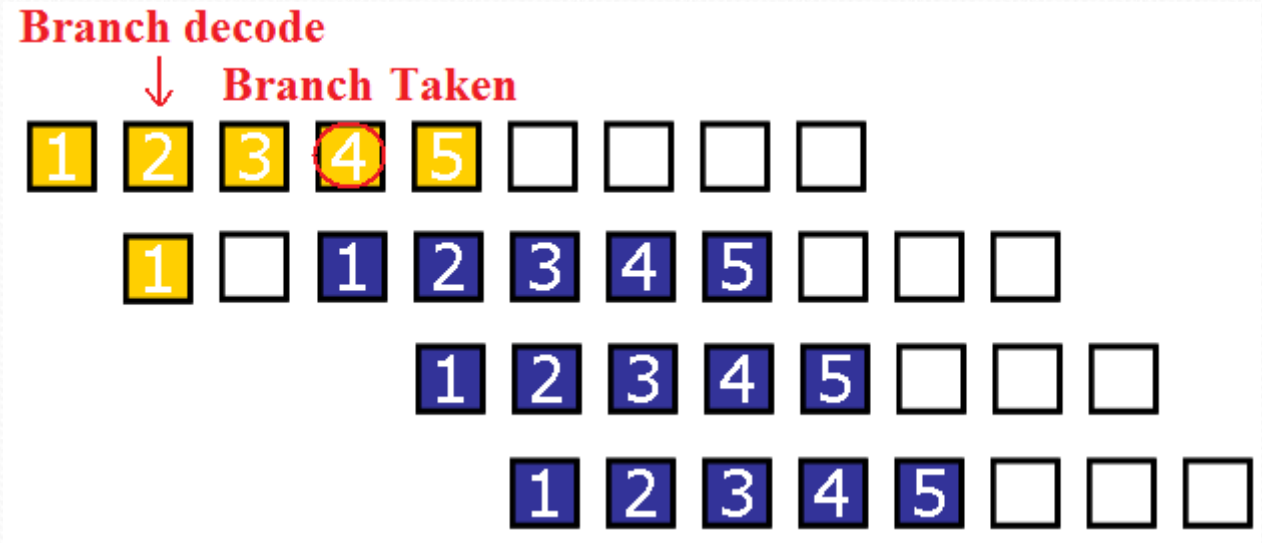
- An alternative scheme is to treat every branch as taken

- As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing the target

Branch (Predicted-taken)



Branch (Predicted-taken)



Delayed Branch

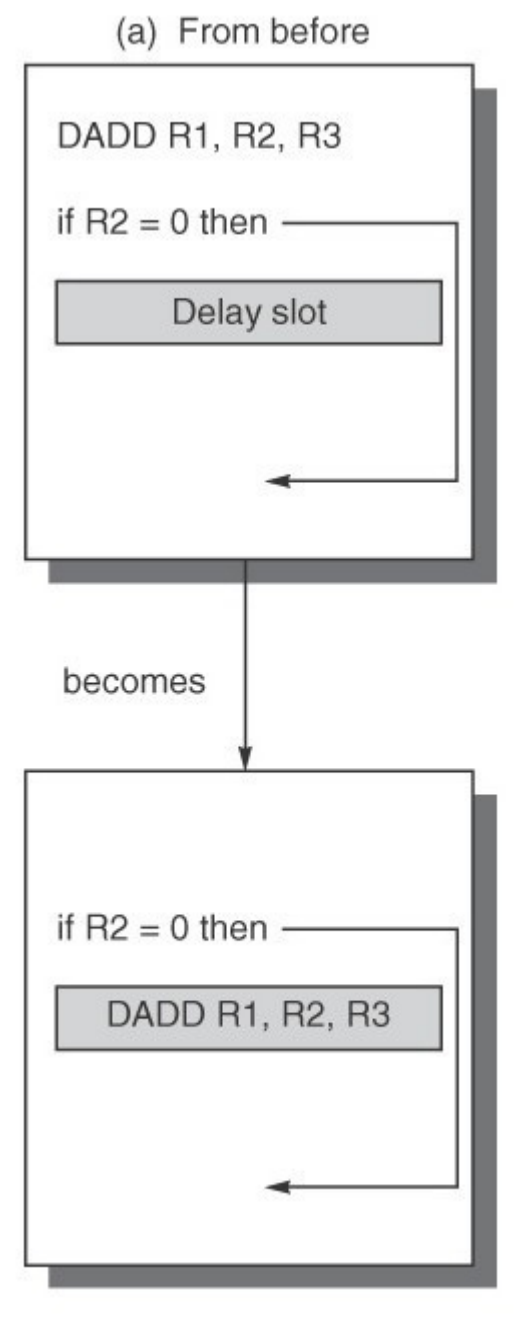
- A fourth scheme in use in some processors is called *delayed branch*
- **It is done in compiler time. It modifies the code**

- *The general format is:*

branch instruction
Delay slot
branch target if taken

Delayed Branch

- Optimal

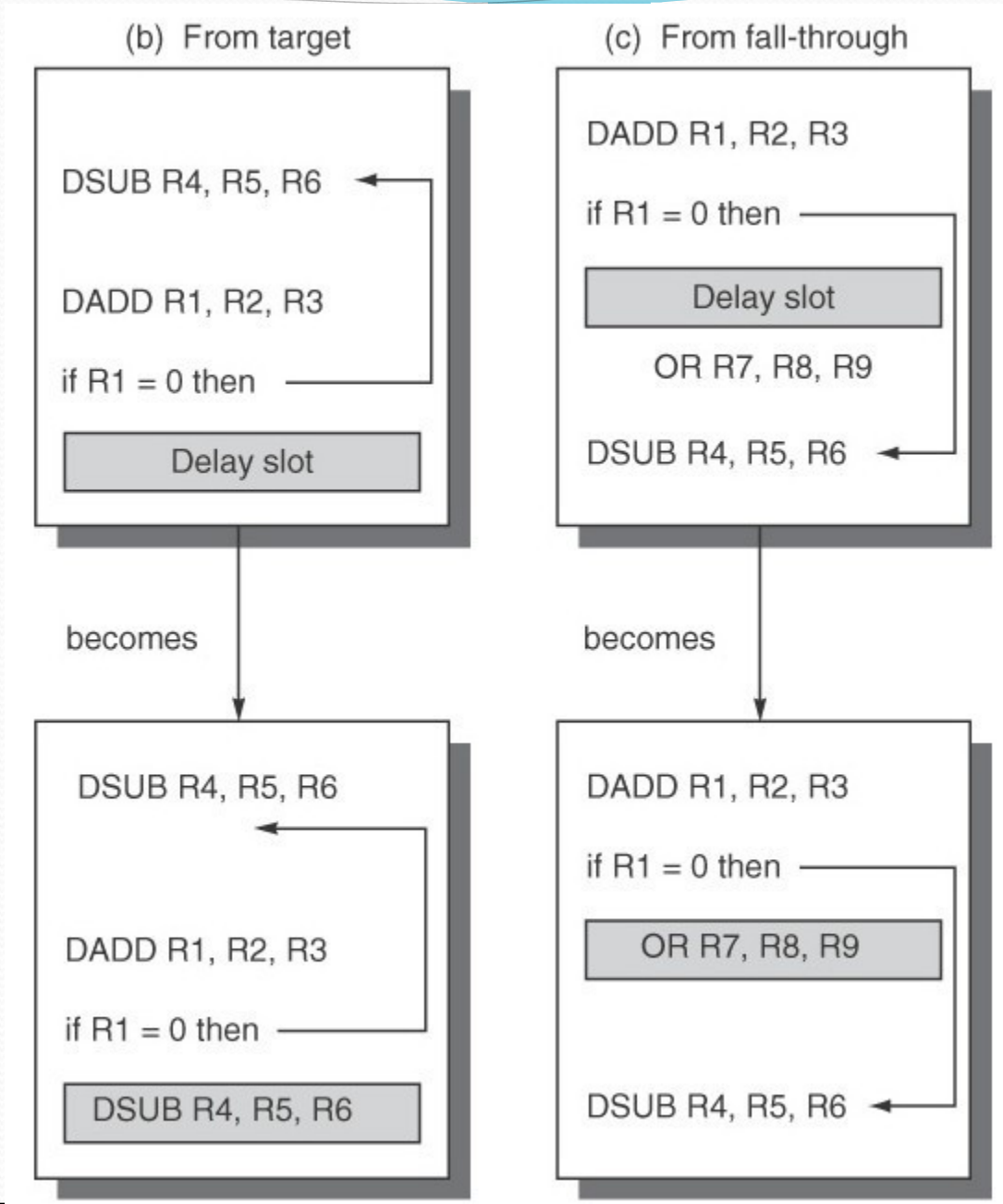


Delaye Branch

If the optimal is not available:

(b)Act like predict-taken (in complier way)

(c)Act like predict-untaken (in complier way)



Delayed Branch

- Delayed Branch is limited by
  - **(1)the restrictions on the instructions that are scheduled into the delay slots (for example: another branch cannot be scheduled)**
  - **(2)our ability to predict at compile time whether a branch is likely to be taken or not**

Branch Prediction

- A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed

Branch Prediction

- Various techniques can be used to predict whether a branch will be taken or not:
  - **Prediction never taken**
  - **Prediction always taken**
  - **Prediction by opcode**
  - **Branch history table**
- The first three approaches are static: they do not depend on the execution history up to the time of the conditional branch instruction. The last approach is dynamic: they depend on the execution history.

Important Pipeline Characteristics

- **Latency**
  - Time required for an instruction to propagate through the pipeline
  - Based on the Number of Stages * Cycle Time. Dominant if there are lots of exceptions/ hazards, i.e. we have to constantly be re-filling the pipeline
- **Throughput**
  - The rate at which instructions can start and finish Dominant if there are few exceptions and hazards, i.e. the pipeline stays mostly full
- **Note we need an increased memory bandwidth over the non-pipelined processor**

Exceptions

- An exception is when the normal execution order of instructions is changed. This has many names:
  - Interrupt
  - Fault
  - Exception
- Examples:
  - request
  - Invoking OS
  - service Page
  - Fault
  - Malfunction
  - Undefined

Eliminating hazards- pubbli Pipeline ng

- *Bub b ling the p ip e line* **, also known as a** *p ip e line b re a k* **or a** *p ip e line s ta ll***, is a method for preventing data, structural, and branch hazards from occurring.**
- **instructions are fetched, control logic determines whether a hazard could/will occur. If this is true, then the control logic inserts NOPs into the pipeline. Thus, before the next instruction (which would cause the**

No: of NOPs = stages in pipeline

- If the number of NOPs is equal to the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards. All forms of stalling introduce a delay before the processor can resume execution.