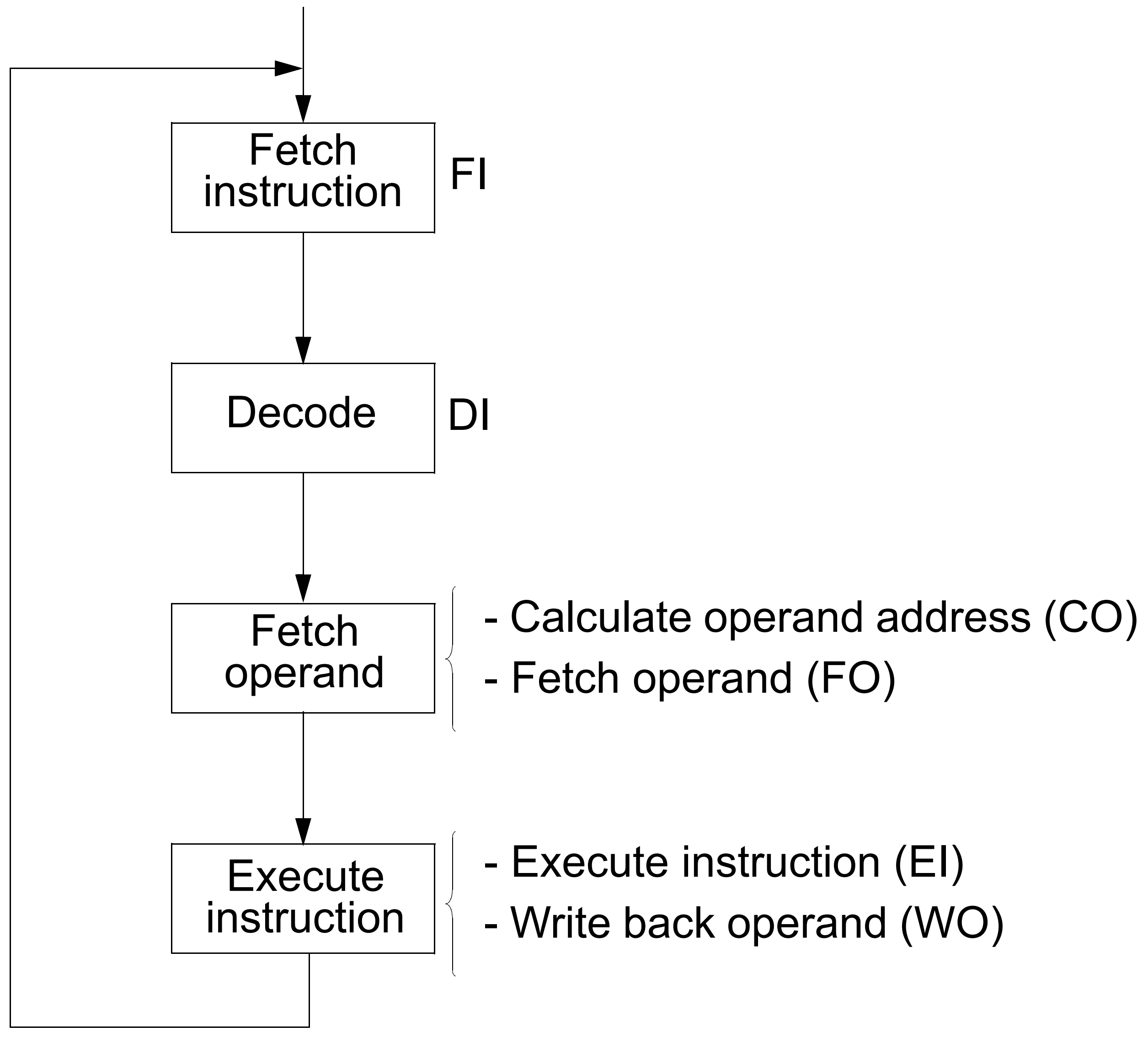
# INSTRUCTION PIPELINING

1. The Instruction Cycle
2. Instruction Pipelining
3. Pipeline Hazards
4. Reducing Branch Penalties
5. Static Branch Prediction
6. Dynamic Branch Prediction
7. Branch History Table

# The Instruction Cycle


Fetch instruction
FI
Decode
DI
Fetch operand
- Calculate operand address (CO)
- Fetch operand (FO)
Execute instruction
- Execute instruction (EI)
- Write back operand (WO)

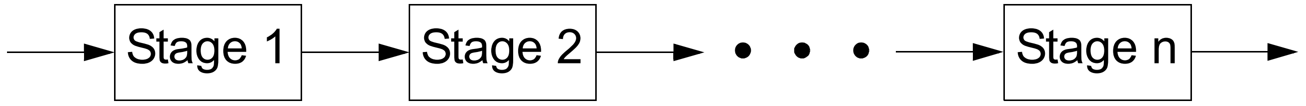
## Instruction Pipelining

- **Instruction execution is extremely complex and involves several operations which are executed *successively*. This implies a large amount of hardware, *but only one part of this hardware works at a given moment*.**

- **Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. This is solved without additional hardware, only letting different parts of the hardware work for different instructions at the same time.**

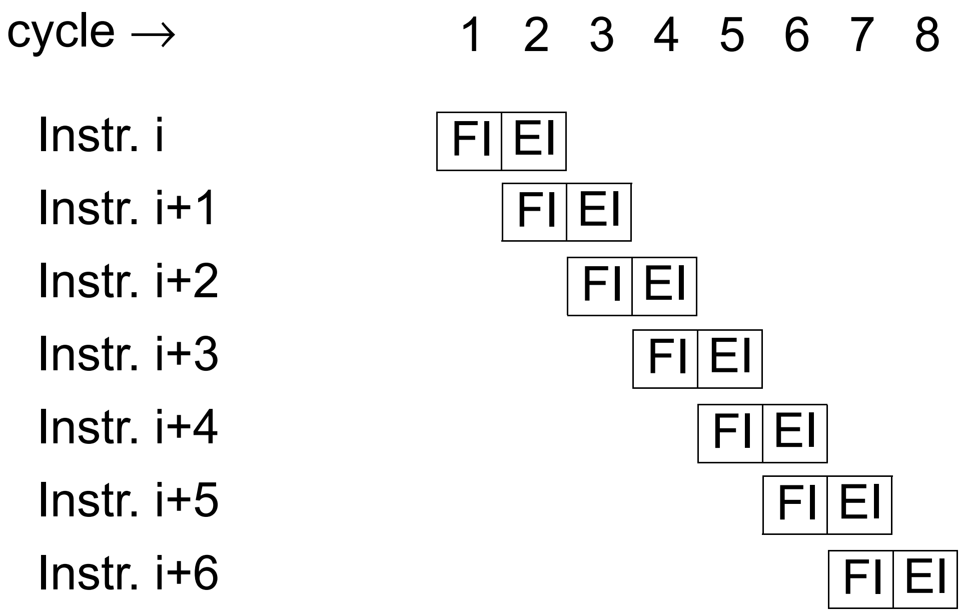
- **The pipeline organization of a CPU is similar to an assembly line: the work to be done in an instruction is broken into smaller steps (pieces), each of which takes a fraction of the time needed to complete the entire instruction. Each of these steps is a *pipe stage* (or a *pipe segment*).**

- **Pipe stages are connected to form a pipe:**



# Acceleration by Pipelining

**Two stage pipeline:** **FI: fetch instruction**
**EI: execute instruction**



**We consider that each instruction takes execution time $T_{ex}$.**

**Execution time for the 7 instructions, with pipelining: $(T_{ex}/2)\times 8 = 4\times T_{ex}$**

- **Acceleration: $7\times T_{ex} / 4\times T_{ex} = 7/4$**

## Acceleration by Pipelining

<u>**Six stage pipeline**</u>**:**

**FI: fetch instruction**
**DI: decode instruction**
**CO: calculate operand address**
**FO: fetch operand**
**EI: execute instruction**
**WO: write operand**

| cycle $\rightarrow$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instr. i | FI | DI | CO | FO | EI | WO | | | | | | |
| Instr. i+1 | | FI | DI | CO | FO | EI | WO | | | | | |
| Instr. i+2 | | | FI | DI | CO | FO | EI | WO | | | | |
| Instr. i+3 | | | | FI | DI | CO | FO | EI | WO | | | |
| Instr. i+4 | | | | | FI | DI | CO | FO | EI | WO | | |
| Instr. i+5 | | | | | | FI | DI | CO | FO | EI | WO | |
| Instr. i+6 | | | | | | | FI | DI | CO | FO | EI | WO |

**Execution time for the 7 instructions, with pipelining: $(T_{ex}/6)\times 12 = 2\times T_{ex}$**

# Acceleration by Pipelining

**Six stage pipeline:**

**FI: fetch instruction**
**DI: decode instruction**
**CO: calculate operand address**
**FO: fetch operand**
**EI: execute instruction**
**WO: write operand**

| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instr. i | FI | DI | CO | FO | EI | WO | | | | | | |
| Instr. i+1 | | FI | DI | CO | FO | EI | WO | | | | | |
| Instr. i+2 | | | FI | DI | CO | FO | EI | WO | | | | |
| Instr. i+3 | | | | FI | DI | CO | FO | EI | WO | | | |
| Instr. i+4 | | | | | FI | DI | CO | FO | EI | WO | | |
| Instr. i+5 | | | | | | FI | DI | CO | FO | EI | WO | |
| Instr. i+6 | | | | | | | FI | DI | CO | FO | EI | WO |

**Execution time for the 7 instructions, with pipelining: $(T_{ex}/6)\times12 = 2\times T_{ex}$**

- **Acceleration: $7\times T_{ex} / 2\times T_{ex} = 7/2$**

- **After a certain time (N-1 cycles) all the N stages of the pipeline are working: the pipeline is filled. Now, *theoretically*, the pipeline works providing maximal parallelism (N instructions are active simultaneously).**

## Acceleration by Pipelining

- $\tau$: duration of one cycle
- *n*: number of instructions to execute
- *k*: number of pipeline stages
- $T_{k,n}$: total time to execute *n* instructions on a pipeline with *k* stages
- $S_{k,n}$: (theoretical) speedup produced by a pipeline with *k* stages when executing *n* instructions

## Acceleration by Pipelining

- $\tau$: duration of one cycle
- $n$: number of instructions to execute
- $k$: number of pipeline stages
- $T_{k,n}$: total time to execute $n$ instructions on a pipeline with $k$ stages
- $S_{k,n}$: (theoretical) speedup produced by a pipeline with $k$ stages when executing $n$ instructions

$$T_{k,n} = [k + (n-1)] \times \tau$$

- The first instruction takes $k \times \tau$ to finish
- The following $n - 1$ instructions produce one result per cycle.

On a non-pipelined processor each instruction takes $k \times \tau$, and $n$ instructions take $T_n = n \times k \times \tau$

$$S_{k,n} = \frac{T_n}{T_{k,n}} = \frac{n \times k \times \tau}{[k + (n-1)] \times \tau} = \frac{n \times k}{k + (n-1)}$$

For large number of instructions ($n \rightarrow \infty$) the speedup approaches $k$ (nr. of stages).

## Acceleration by Pipelining

- **Apparently a greater number of stages always provides better performance.**

  **However:**

  - **a greater number of stages increases the overhead in moving information between stages and synchronization between stages.**
  - **with the number of stages the complexity of the CPU grows.**
  - **it is difficult to keep a large pipeline at maximum rate because of *pipeline hazards*.**

## Acceleration by Pipelining

- Apparently a greater number of stages always provides better performance.

  However:

  - a greater number of stages increases the overhead in moving information between stages and synchronization between stages.
  - with the number of stages the complexity of the CPU grows.
  - it is difficult to keep a large pipeline at maximum rate because of *pipeline hazards*.

<u>80486 and Pentium</u>: five-stage pipeline for integer instructions
eight-stage pipeline for FP instructions

<u>PowerPC</u>: four-stage pipeline for integer instructions
six-stage pipeline for FP instructions

## Pipeline Hazards

- Pipeline hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle. The instruction is said to be *stalled*.
  - When an instruction is stalled, all instructions later in the pipeline than the stalled instruction are also stalled. Instructions earlier than the stalled one can continue. No new instructions are fetched during the stall.

## Pipeline Hazards

- Pipeline hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle. The instruction is said to be *stalled*.
  - When an instruction is stalled, all instructions later in the pipeline than the stalled instruction are also stalled. Instructions earlier than the stalled one can continue. No new instructions are fetched during the stall.

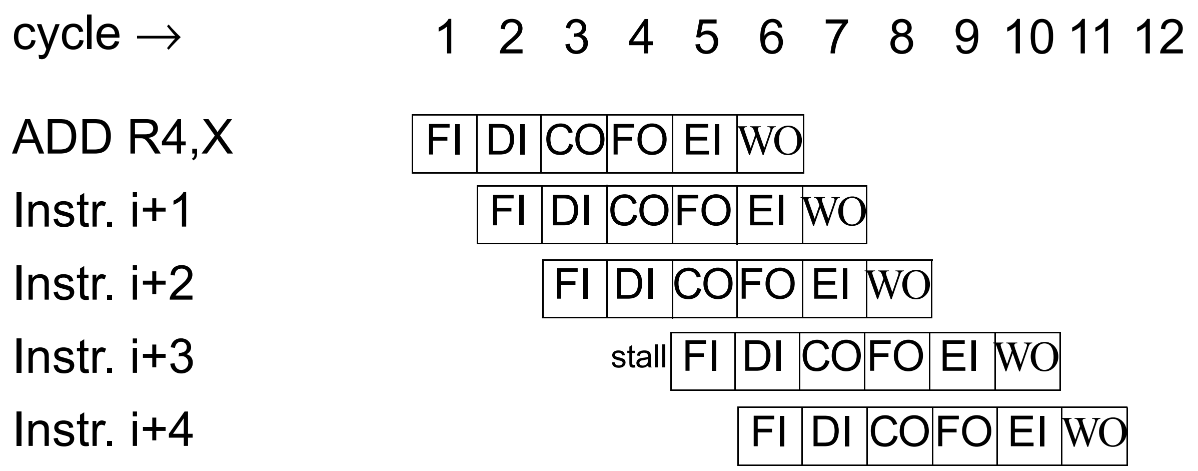
- Types of hazards:

  1. Structural hazards
  2. Data hazards
  3. Control hazards

# Structural Hazards

- **Structural hazards occur when a certain resource (memory, functional unit) is requested by more than one instruction at the same time.**

**Consider Instruction ADD R4,X**

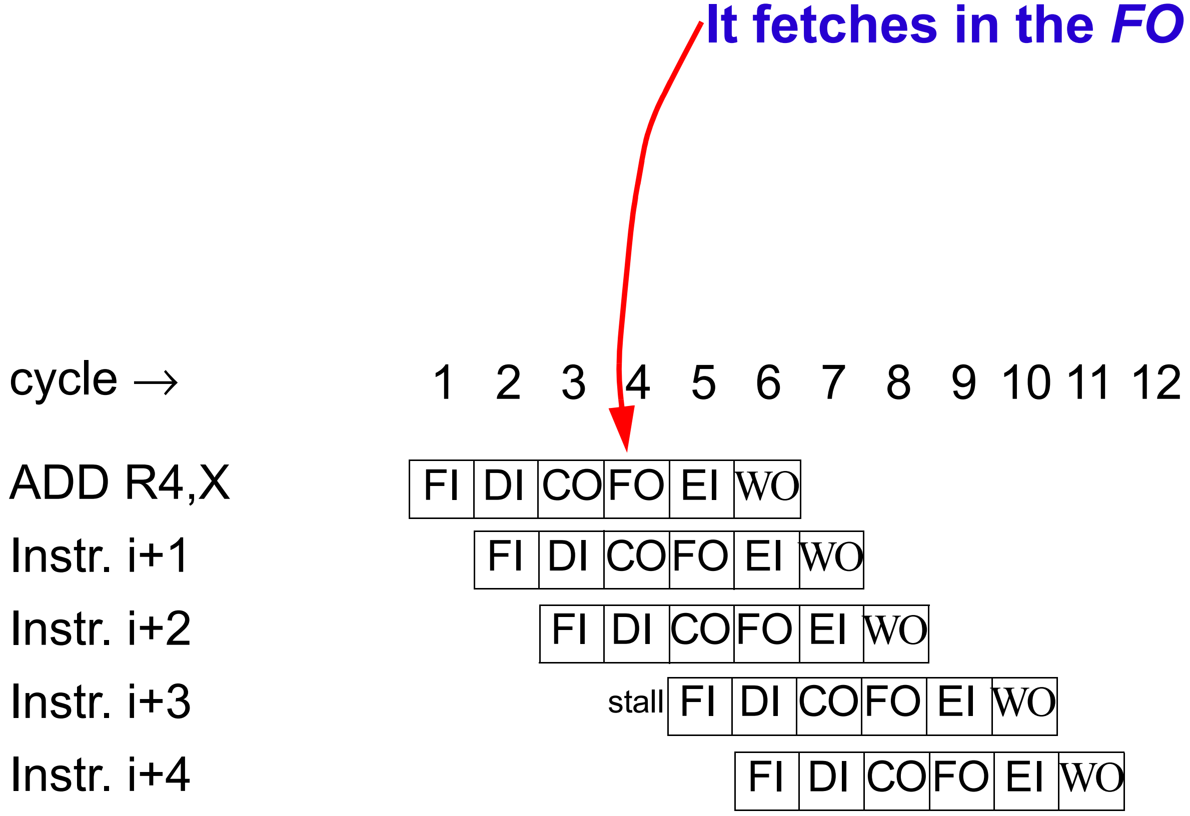


# Structural Hazards

- **Structural hazards occur when a certain resource (memory, functional unit) is requested by more than one instruction at the same time.**

**<u>Consider Instruction</u> ADD R4,X**

**It fetches in the *FO* stage operand X from memory.**



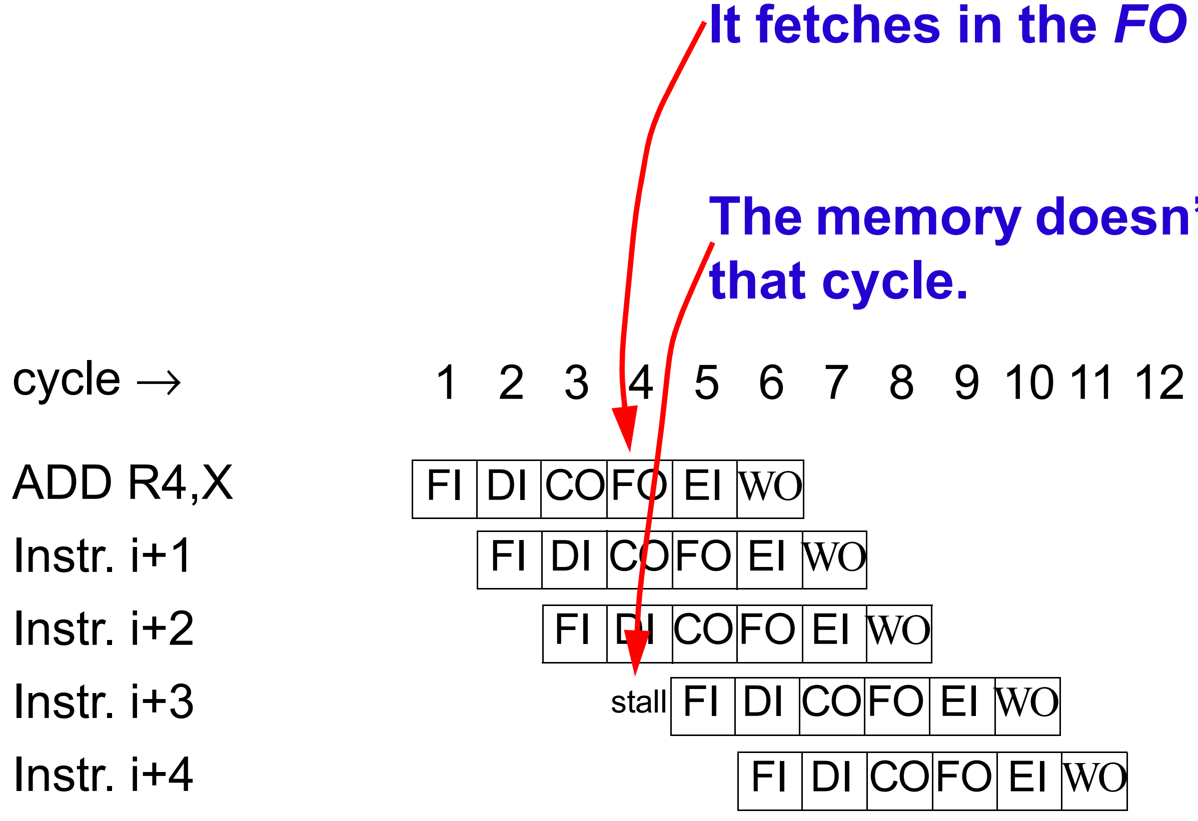
# Structural Hazards

- Structural hazards occur when a certain resource (memory, functional unit) is requested by more than one instruction at the same time.

<u>Consider Instruction</u> ADD R4,X

It fetches in the *FO* stage operand X from memory.

The memory doesn't accept another access during that cycle.



| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD R4,X | FI | DI | CO | FO | EI | WO | | | | | | |
| Instr. i+1 | | FI | DI | CO | FO | EI | WO | | | | | |
| Instr. i+2 | | | FI | DI | CO | FO | EI | WO | | | | |
| Instr. i+3 | | | | stall | FI | DI | CO | FO | EI | WO | | |
| Instr. i+4 | | | | | | FI | DI | CO | FO | EI | WO | |

<u>Penalty</u>: 1 cycle

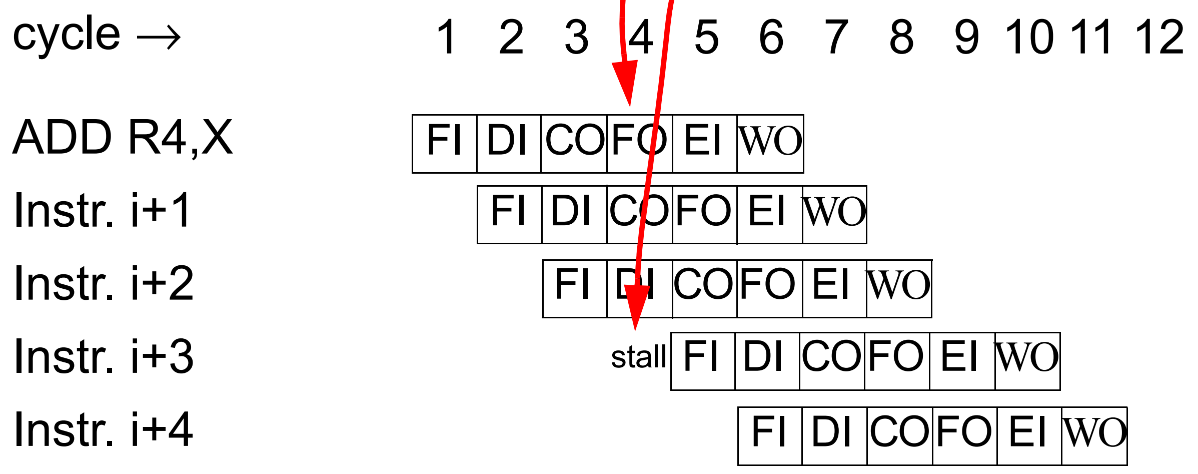
# Structural Hazards

- **Structural hazards occur when a certain resource (memory, functional unit) is requested by more than one instruction at the same time.**

**<u>Consider Instruction</u> ADD R4,X**

**It fetches in the *FO* stage operand X from memory.**

**The memory doesn’t accept another access during that cycle.**



| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD R4,X | FI | DI | CO | FO | EI | WO | | | | | | |
| Instr. i+1 | | FI | DI | CO | FO | EI | WO | | | | | |
| Instr. i+2 | | | FI | DI | CO | FO | EI | WO | | | | |
| Instr. i+3 | | | | stall | FI | DI | CO | FO | EI | WO | | |
| Instr. i+4 | | | | | | FI | DI | CO | FO | EI | WO | |

**<u>Penalty</u>: 1 cycle**

**Certain resources are duplicated in order to avoid structural hazards. Functional units (ALU, FP unit) can be pipelined themselves in order to support several instructions at a time. *A classical way to avoid hazards at memory access is by providing separate data and instruction caches.***

## Data Hazards

- We have two instructions, I1 and I2. The execution of I2 starts before I1 has terminated. If I2 needs the result produced by I1, but this result has not yet been generated, we have a data hazard.

| | | |
|---|---|---|
| I1: | MUL R2,R3 | $R2 \leftarrow R2 * R3$ |
| I2: | ADD R1,R2 | $R1 \leftarrow R1 + R2$ |

# Data Hazards

- **We have two instructions, I1 and I2. The execution of I2 starts before I1 has terminated. If I2 needs the result produced by I1, but this result has not yet been generated, we have a data hazard.**

**I1: MUL R2,R3 $\qquad$ R2 $\leftarrow$ R2 * R3**
**I2: ADD R1,R2 $\qquad$ R1 $\leftarrow$ R1 + R2**

| cycle $\rightarrow$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MUL R2,R3 | FI | DI | CO | FO | EI | WO | | | | | | |
| ADD R1,R2 | | FI | DI | CO | stall | stall | FO | EI | WO | | | |
| Instr. i+2 | | | FI | DI | | | CO | FO | EI | WO | | |

**Before executing its FO stage, the ADD instruction is stalled until the MUL instruction has written the result into R2.**

**Penalty: 2 cycles**

## Data Hazards

- Some of the penalty produced by data hazards can be avoided using a technique called *forwarding* (bypassing).

    - If the hardware detects that the value needed for the current operation is the one produced by the ALU in the previous operation (but which has not yet been written back) it uses directly the value from the output of the ALU, instead of waiting that the result is written back to the register.

## Data Hazards

**Our previous example**

| | | |
|---|---|---|
| I1: | MUL R2,R3 | $R2 \leftarrow R2 * R3$ |
| I2: | ADD R1,R2 | $R1 \leftarrow R1 + R2$ |

# Data Hazards

## Our previous example

**I1: MUL R2,R3** **$R2 \leftarrow R2 * R3$**
**I2: ADD R1,R2** **$R1 \leftarrow R1 + R2$**

## Without forwarding:

**Penalty: 2 cycles**

| cycle $\rightarrow$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MUL R2,R3 | FI | DI | CO | FO | EI | WO | | | | | | |
| ADD R1,R2 | | FI | DI | CO | stall | stall | FO | EI | WO | | | |
| Instr. i+2 | | | FI | DI | | | CO | FO | EI | WO | | |

# Data Hazards

## Our previous example

**I1: MUL R2,R3** $\quad R2 \leftarrow R2 * R3$

**I2: ADD R1,R2** $\quad R1 \leftarrow R1 + R2$

## Without forwarding:

**Penalty: 2 cycles**

| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MUL R2,R3 | FI | DI | CO | FO | EI | WO | | | | | | |
| ADD R1,R2 | | FI | DI | CO | stall | stall | FO | EI | WO | | | |
| Instr. i+2 | | | FI | DI | | | CO | FO | EI | WO | | |

## With forwarding

**Penalty: 1 cycle**

| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MUL R2,R3 | FI | DI | CO | FO | EI | WO | | | | | | |
| ADD R1,R2 | | FI | DI | CO | stall | FO | EI | WO | | | | |

# Control Hazards

- Control hazards are produced by branch instructions.

## Unconditional branch

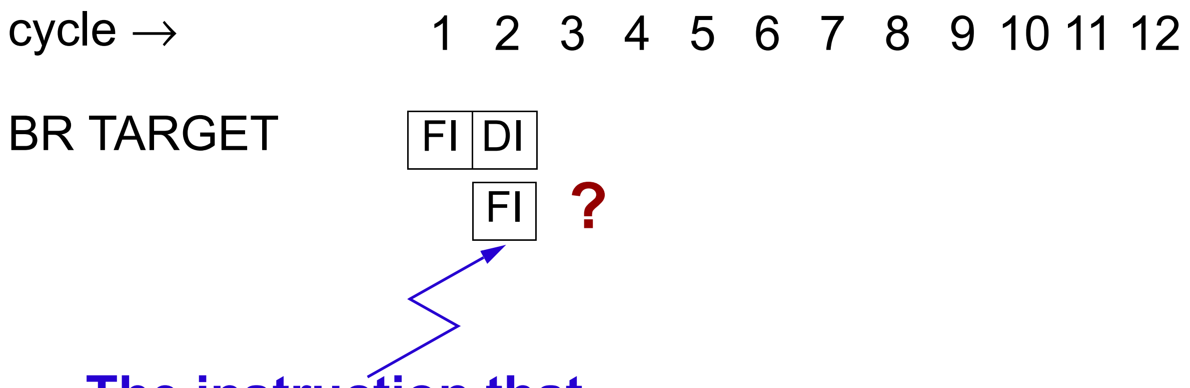
```
                 - - - - - - - - - - - - - - -
                 BR     TARGET
                 - - - - - - - - - - - - - - -
      TARGET     - - - - - - - - - - - - - - -
```

# Control Hazards

- Control hazards are produced by branch instructions.

**Unconditional branch**

```
           - - - - - - - - - - - - - - -
           BR     TARGET
           - - - - - - - - - - - - - - -
TARGET     - - - - - - - - - - - - - - -
```



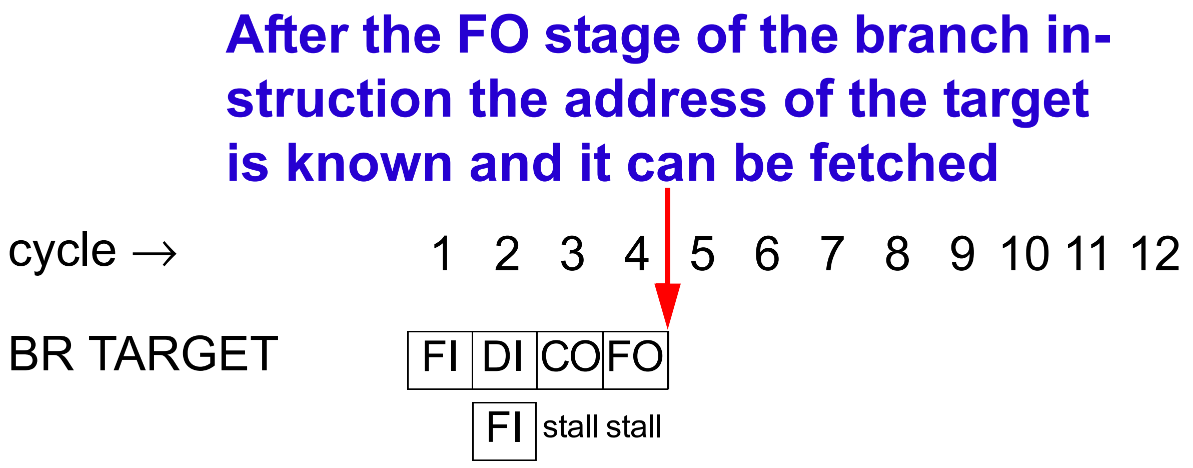
**The instruction that follows BR is fetched!**

## Control Hazards

- Control hazards are produced by branch instructions.

**Unconditional branch**

```
                - - - - - - - - - - - - - - -
                BR    TARGET
                - - - - - - - - - - - - - - -
TARGET          - - - - - - - - - - - - - - -
```


After the FO stage of the branch instruction the address of the target is known and it can be fetched
cycle →
1 2 3 4 5 6 7 8 9 10 11 12
BR TARGET
FI
DI
CO
FO
FI
stall
stall

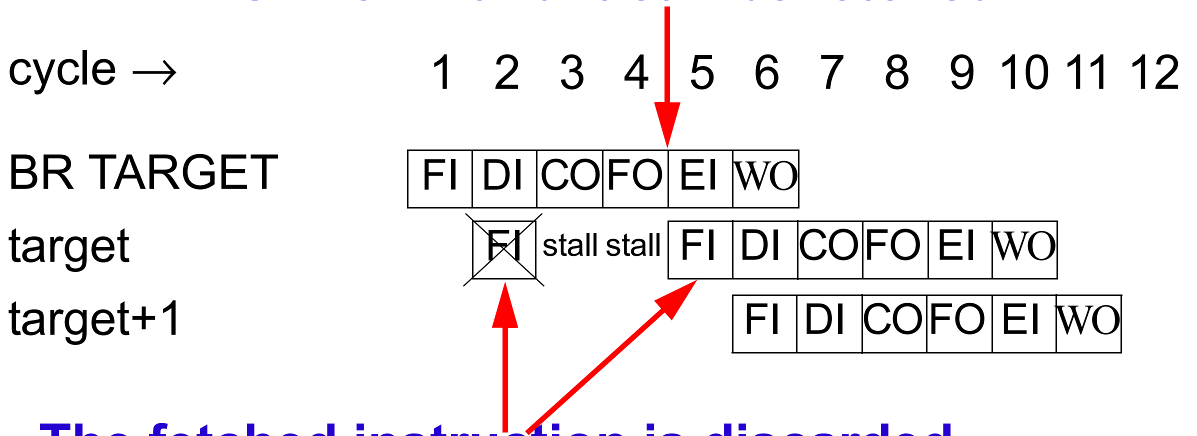
# Control Hazards

- Control hazards are produced by branch instructions.

**Unconditional branch**

```
            - - - - - - - - - - - - - - -
            BR    TARGET
            - - - - - - - - - - - - - - -
TARGET      - - - - - - - - - - - - - - -
```

**After the FO stage of the branch instruction the address of the target is known and it can be fetched**



**Penalty: 3 cycles**

**The fetched instruction is discarded and the target instruction is fetched!**

## Control Hazards

**Conditional branch**

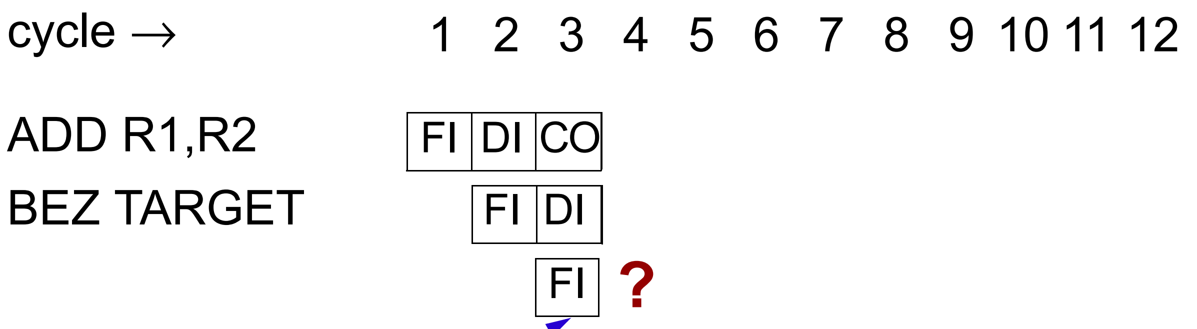
```
                ADD    R1,R2          R1 ← R1 + R2
                BEZ    TARGET         branch if zero
                instruction i+1
                - - - - - - - - - - - - - -
TARGET          - - - - - - - - - - - - - -
```

# Control Hazards

## Conditional branch

```
           ADD    R1,R2          R1 ← R1 + R2
           BEZ    TARGET         branch if zero
           instruction i+1
           - - - - - - - - - - - - - -
TARGET     - - - - - - - - - - - - - -
```

**Branch is taken**



***instruction i+1*** **(that follows BEZ) is fetched!**

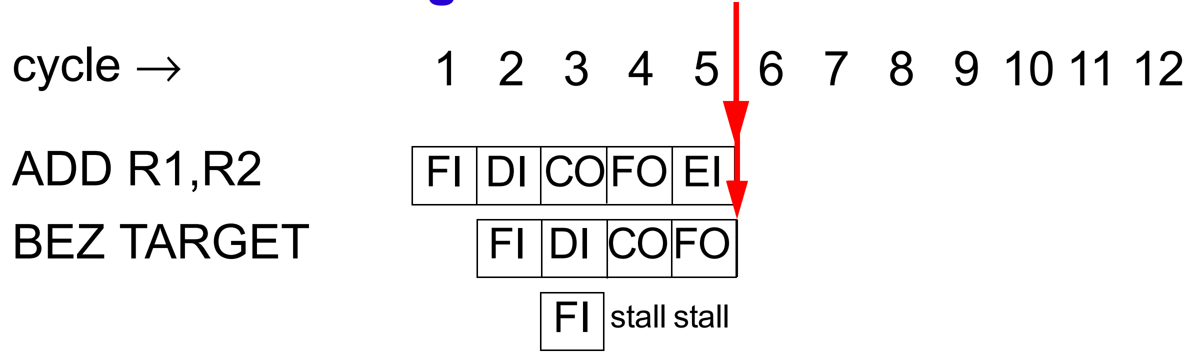
# Control Hazards

## Conditional branch

```
             ADD    R1,R2            R1 ← R1 + R2
             BEZ    TARGET           branch if zero
             instruction i+1
             - - - - - - - - - - - - -
TARGET       - - - - - - - - - - - - -
```

## Branch is taken

At this moment, both the condition (set by ADD) and the target address are known.



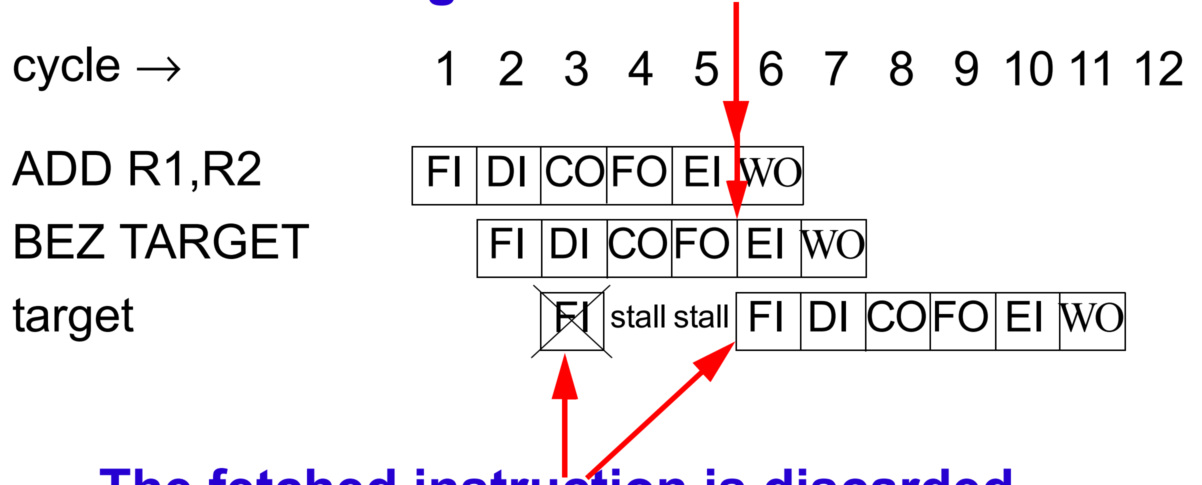
# Control Hazards

## Conditional branch

| | | | |
|---|---|---|---|
| | ADD | R1,R2 | R1 ← R1 + R2 |
| | BEZ | TARGET | branch if zero |
| | instruction i+1 | | |
| | - - - - - - - - - - - - - - | | |
| TARGET | - - - - - - - - - - - - - - | | |

## Branch is taken

**At this moment, both the condition (set by ADD) and the target address are known.**



**Penalty: 3 cycles**

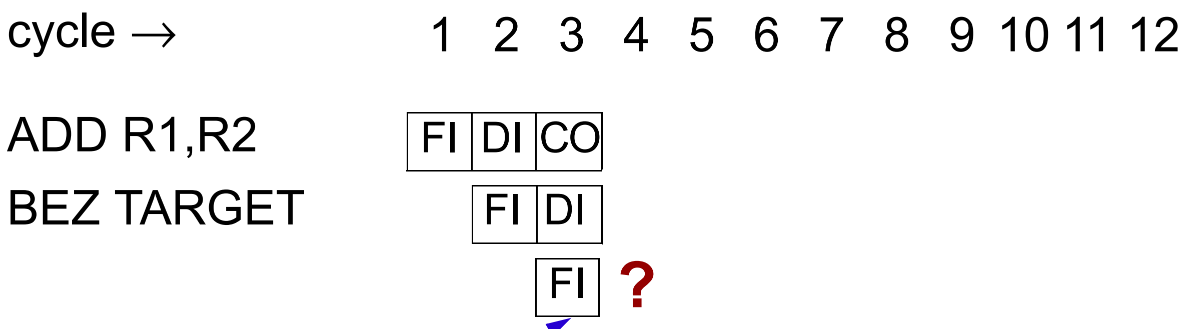
**The fetched instruction is discarded and the target instruction is fetched!**

# Control Hazards

## Conditional branch

```
          ADD    R1,R2          R1 ← R1 + R2
          BEZ    TARGET         branch if zero
          instruction i+1
          - - - - - - - - - - - - - -
TARGET    - - - - - - - - - - - - - -
```

## Branch is not taken



***instruction i+1*** **(that follows BEZ) is fetched!**

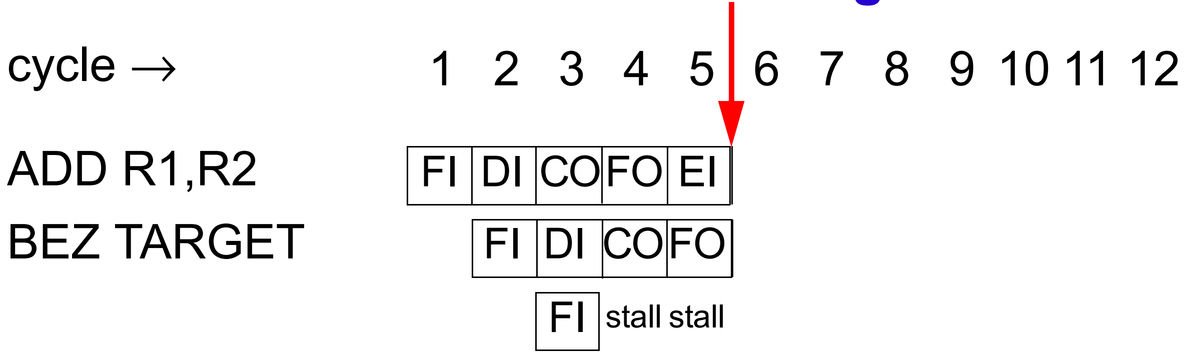
## Control Hazards

**Conditional branch**

```
           ADD     R1,R2           R1 ← R1 + R2
           BEZ     TARGET          branch if zero
           instruction i+1
           - - - - - - - - - - - - - - -
TARGET     - - - - - - - - - - - - - - -
```

**Branch is not taken**

**At this moment the condition (set by ADD) is known and *instruction i+1* can go on.**



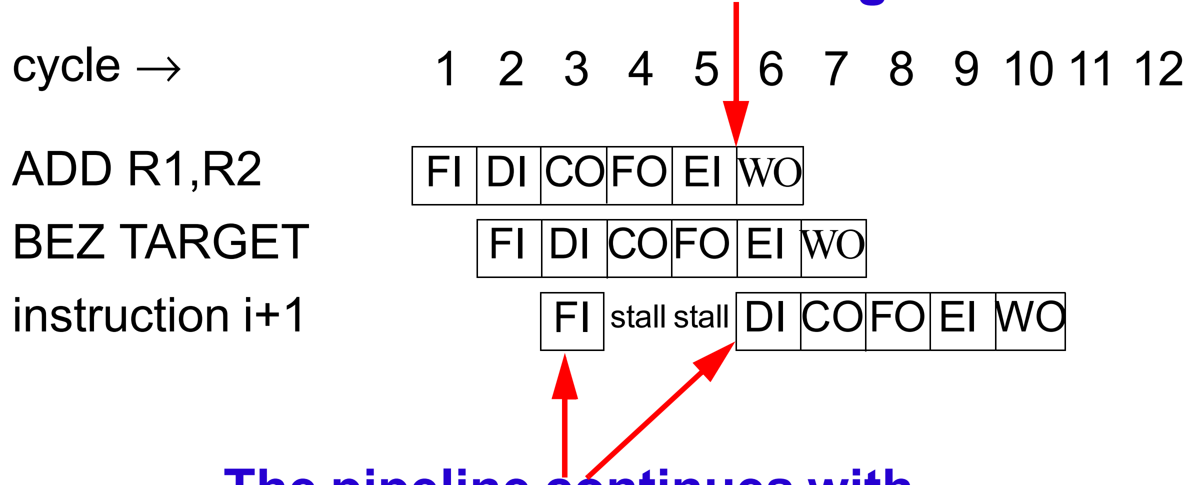
# Control Hazards

## Conditional branch

```
            ADD    R1,R2        R1 ← R1 + R2
            BEZ    TARGET       branch if zero
            instruction i+1
            - - - - - - - - - - - - - -
TARGET      - - - - - - - - - - - - - -
```

## Branch is not taken

At this moment the condition (set by ADD) is known and *instruction i+1* can go on.



**Penalty: 2 cycles**

The pipeline continues with the fetched instruction!

## Control Hazards

- With conditional branch we have a penalty even if the branch has *not* been taken. This is because we have to wait until the branch condition is available.

- Branch instructions represent a major problem in assuring an optimal flow through the pipeline. Several approaches have been taken for reducing branch penalties.

## Reducing Pipeline Branch Penalties

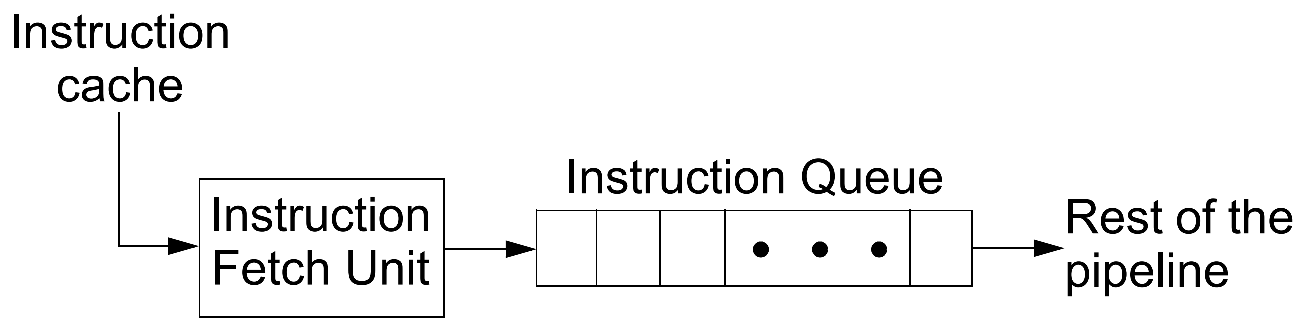
- Branch instructions can dramatically affect pipeline performance. Control operations (conditional and unconditional branch) are very frequent in current programs.

- Some statistics:
  - 20% - 35% of the instructions executed are branches (conditional and unconditional).
  - Conditional branches are much more frequent than unconditional ones (more than two times). More than 50% of conditional branches are taken.

- It is very important to reduce the penalties produced by branches.

# Instruction Fetch Unit and Instruction Queue

- **Most processors employ sophisticated fetch units that fetch instructions before they are needed and store them in a queue.**



- **The fetch unit also has the ability to recognize branch instructions and to generate the target address.**

**<u>The penalty produced by *unconditional branches* can be drastically reduced</u>: the fetch unit computes the target address and continues to fetch instructions from that address, which are sent to the queue. Thus, the rest of the pipeline gets a continuous stream of instructions, without stalling.**

## Instruction Fetch Unit and Instruction Queue

- **The rate at which instructions can be read (from the instruction cache) must be sufficiently high to avoid an empty queue.**

- **With *conditional branches* penalties can not be avoided. The branch condition, which usually depends on the result of the preceding instruction, has to be known in order to determine the following instruction.**

**Observation**

**In the Pentium 4, the instruction cache (trace cache) is located between the fetch unit and the instruction queue (See lecture on cache memory).**

# Delayed Branching

**The pipeline sequences for a conditional branch instruction:**

**Branch is taken**

**At this moment, both the condition (set by ADD) and the target address are known.**

| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD R1,R2 | FI | DI | CO | FO | EI | WO | | | | | | |
| BEZ TARGET | | FI | DI | CO | FO | EI | WO | | | | | |
| target | | | ~~FI~~ | stall | stall | FI | DI | CO | FO | EI | WO | |

**Penalty: 3 cycles**

**Branch is not taken**

**At this moment the condition (set by ADD) is known and *instruction i+1* can go on.**

| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD R1,R2 | FI | DI | CO | FO | EI | WO | | | | | | |
| BEZ TARGET | | FI | DI | CO | FO | EI | WO | | | | | |
| instruction i+1 | | | FI | stall | stall | DI | CO | FO | EI | WO | | |

**Penalty: 2 cycles**

# Delayed Branching

**The pipeline sequences for a conditional branch instruction:**

## Branch is taken

**At this moment, both the condition (set by ADD) and the target address are known.**

| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD R1,R2 | FI | DI | CO | FO | EI | WO | | | | | | |
| BEZ TARGET | | FI | DI | CO | FO | EI | WO | | | | | |
| target | | | ~~FI~~ | stall | stall | FI | DI | CO | FO | EI | WO | |

**Penalty: 3 cycles**

## Branch is not taken

**At this moment the condition (set by ADD) is known and *instruction i+1* can go on.**

| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD R1,R2 | FI | DI | CO | FO | EI | WO | | | | | | |
| BEZ TARGET | | FI | DI | CO | FO | EI | WO | | | | | |
| instruction i+1 | | | FI | stall | stall | DI | CO | FO | EI | WO | | |

**Penalty: 2 cycles**

**Let the machine do something useful during this time!**

## Delayed Branching

- The idea with delayed branching is to let the CPU do some useful work during some of the cycles which are shown above to be stalled.

- With delayed branching the CPU always executes the instruction that immediately follows after the branch and only then alters (if necessary) the sequence of execution. The instruction after the branch is said to be in the *branch delay slot*.

## Delayed Branching

**This code is produced by a compiler, for a machine *without delayed branching*:**

**This instruction should be executed only if the branch is not taken.** $\rightarrow$ MOVE

```
        MUL    R3,R4     R3 ← R3*R4
        SUB    #1,R2     R2 ← R2-1
        ADD    R1,R2     R1 ← R1+R2
        BEZ    TAR       branch if zero
        MOVE   #10,R1    R1 ← 10
        - - - - - - - - - - - - - -
TAR     - - - - - - - - - - - - - -
```

- **The compiler (assembler) has to find an instruction which can be moved from its original place into *the* branch delay slot after the branch and which will be executed regardless of the outcome of the branch.**

# Delayed Branching

<u>This code is produced by a compiler, for a machine *without delayed branching*:</u>

**Doesn’t influence any of the following instructions until the branch; also doesn’t influence the outcome of the branch.**

```
        MUL     R3,R4      R3 ← R3*R4
        SUB     #1,R2      R2 ← R2-1
        ADD     R1,R2      R1 ← R1+R2
        BEZ     TAR        branch if zero
        MOVE    #10,R1     R1 ← 10
        - - - - - - - - - - - - -
TAR     - - - - - - - - - - - - -
```

- **The compiler (assembler) has to find an instruction which can be moved from its original place into *the* branch delay slot after the branch and which will be executed regardless of the outcome of the branch.**

# Delayed Branching

**This code is produced by a compiler, for a machine *without delayed branching*:**

**Doesn't influence any of the following instructions until the branch; also doesn't influence the outcome of the branch.**

```
       MUL     R3,R4     R3 ← R3*R4
       SUB     #1,R2     R2 ← R2-1
       ADD     R1,R2     R1 ← R1+R2
       BEZ     TAR       branch if zero
       MOVE    #10,R1    R1 ← 10
       - - - - - - - - - - - - - -
TAR    - - - - - - - - - - - - - -
```

**This code is produced by a compiler, for a machine *with delayed branching*:**

```
       SUB     #1,R2     R2 ← R2-1
       ADD     R1,R2     R1 ← R1+R2
       BEZ     TAR       branch if zero
       MUL     R3,R4     R3 ← R3*R4
       MOVE    #10,R1    R1 ← 10
       - - - - - - - - - - - - - -
TAR    - - - - - - - - - - - - - -
```

**Executed regardless of the condition.** (MUL)

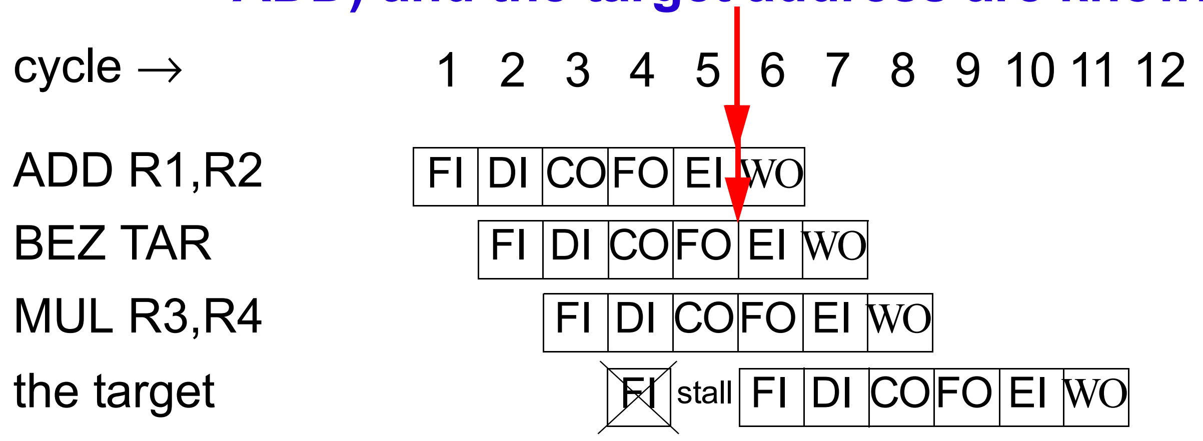
**Executed only if branch *not* taken.** (MOVE)

# Delayed Branching

**The pipeline sequences with delayed branching:**

## Branch is taken

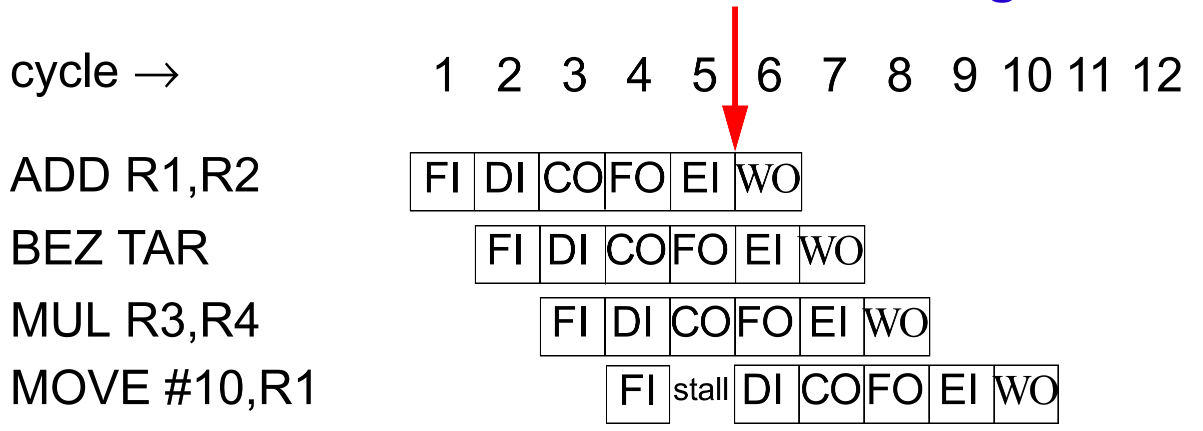
**At this moment the condition (set by ADD) and the target address are known.**



**Penalty: 2 cycles**

## Branch is not taken

**At this moment the condition is known and the MOVE can go on.**



**Penalty: 1 cycle**

## Delayed Branching

- **What happens if the compiler is not able to find an instruction to be moved after the branch, into the branch delay slot?**

# Delayed Branching

- What happens if the compiler is not able to find an instruction to be moved after the branch, into the branch delay slot?

  In this case a NOP instruction (an instruction that does nothing) has to be placed after the branch. In this case the penalty will be the same as without delayed branching.

```
        MUL     R2,R4
        SUB     #1,R2
        ADD     R1,R2
        BEZ     TAR
        NOP
        MOVE    #10,R1
        - - - - - - - - - - - - - -
TAR     - - - - - - - - - - - - - -
```

Now, with R2, this instruction influences the following ones and cannot be moved from its place.

- Some statistics show that for between 60% and 85% of branches, sophisticated compilers are able to find an instruction to be moved into the branch delay slot.

# Branch Prediction

- In the last example we have considered (predicted) that the *branch will not be taken* and we fetched the instruction following the branch; in the case the branch was taken the fetched instruction was discarded. As result, we had:

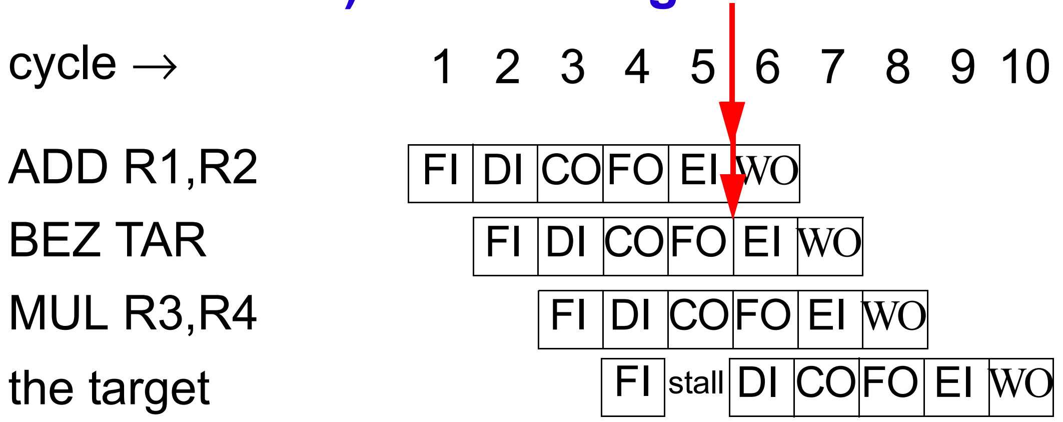
$$\begin{tikzcd} & \text{1 if the branch is not taken (prediction fulfilled)} \\ \text{branch penalty of} \arrow[ur] \arrow[dr] & \\ & \text{2 if the branch is taken (prediction \textit{not} fulfilled)} \end{tikzcd}$$

# Branch Prediction

- Let us consider the opposite prediction: *branch taken*. For this solution it is needed that the target address is computed in advance by an instruction fetch unit.

<u>Branch is taken</u>

At this moment the condition (set by ADD) and the target address are known.

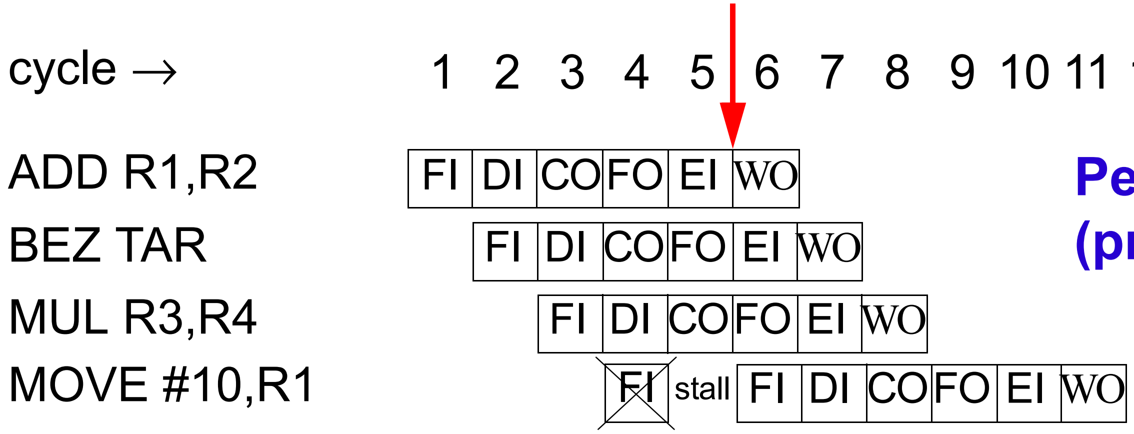


| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD R1,R2 | FI | DI | CO | FO | EI | WO | | | | | | |
| BEZ TAR | | FI | DI | CO | FO | EI | WO | | | | | |
| MUL R3,R4 | | | FI | DI | CO | FO | EI | WO | | | | |
| the target | | | | FI | stall | DI | CO | FO | EI | WO | | |

Penalty: 1 cycle (prediction fulfilled)

<u>Branch is not taken</u>

At this moment the condition is known and the MOVE can go on.



| cycle → | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ADD R1,R2 | FI | DI | CO | FO | EI | WO | | | | | | |
| BEZ TAR | | FI | DI | CO | FO | EI | WO | | | | | |
| MUL R3,R4 | | | FI | DI | CO | FO | EI | WO | | | | |
| MOVE #10,R1 | | | | ~~FI~~ | stall | FI | DI | CO | FO | EI | WO | |

Penalty: 2 cycles (prediction *not* fulfilled)

# Branch Prediction

- Correct branch prediction is very important and can produce substantial performance improvements.
- Based on the predicted outcome, the respective instruction can be fetched, as well as the instructions following it, and they can be placed into the instruction queue. If, after the branch condition is computed, it turns out that the prediction was correct, execution continues. On the other hand, if the prediction is not fulfilled, the fetched instruction(s) must be discarded and the correct instruction must be fetched.

## Branch Prediction

- Correct branch prediction is very important and can produce substantial performance improvements.

- Based on the predicted outcome, the respective instruction can be fetched, as well as the instructions following it, and they can be placed into the instruction queue. If, after the branch condition is computed, it turns out that the prediction was correct, execution continues. On the other hand, if the prediction is not fulfilled, the fetched instruction(s) must be discarded and the correct instruction must be fetched.

- To take full advantage of branch prediction, we can have the instructions not only fetched but also begin execution. This is known as *speculative execution*.

- *Speculative execution* means that instructions are executed before the processor is certain that they are in the correct execution path. If it turns out that the prediction was correct, execution goes on without introducing any branch penalty. If, however, the prediction is not fulfilled, the instruction(s) started in advance and all their associated data must be purged and the state previous to their execution restored.

## Branch Prediction

**Branch prediction strategies**:

1. Static prediction
2. Dynamic prediction

# Static Branch Prediction

- **Static prediction techniques do not take into consideration execution history.**

**Static approaches:**

- **Predict never taken (Motorola 68020): assumes that the branch is not taken.**
- **Predict always taken: assumes that the branch is taken.**
- **Predict depending on the branch direction (PowerPC 601):**
  - **predict branch taken for backward branches;**
  - **predict branch not taken for forward branches.**

# Dynamic Branch Prediction

- Dynamic prediction techniques improve the accuracy of the prediction by recording the history of conditional branches.

# Dynamic Branch Prediction

- Dynamic prediction techniques improve the accuracy of the prediction by recording the history of conditional branches.

## One-Bit Prediction Scheme

- One-bit is used in order to record if the last execution resulted in a branch taken or not. The system predicts the same behavior as for the last time.

# Dynamic Branch Prediction

- Dynamic prediction techniques improve the accuracy of the prediction by recording the history of conditional branches.

## One-Bit Prediction Scheme

- One-bit is used in order to record if the last execution resulted in a branch taken or not. The system predicts the same behavior as for the last time.

Sometimes it does not work so very well:

When a branch is almost always taken, then when it is not taken, we will predict incorrectly twice, rather than once:

```
        - - - - - - - - - - - -
LOOP    - - - - - - - - - - - -
        - - - - - - - - - - - -
        BNZ   LOOP
        - - - - - - - - - - - -
```

After the loop has been executed for the first time and left, it will be remembered that BNZ has not been taken. Now, when the loop is executed again, after the first iteration there will be a false prediction; following predictions are OK until the last iteration, when there will be ***a second false prediction***.

- In this case the result is even worse than with static prediction considering that backward loops are always taken (PowerPC 601 approach).

# Dynamic Branch Prediction

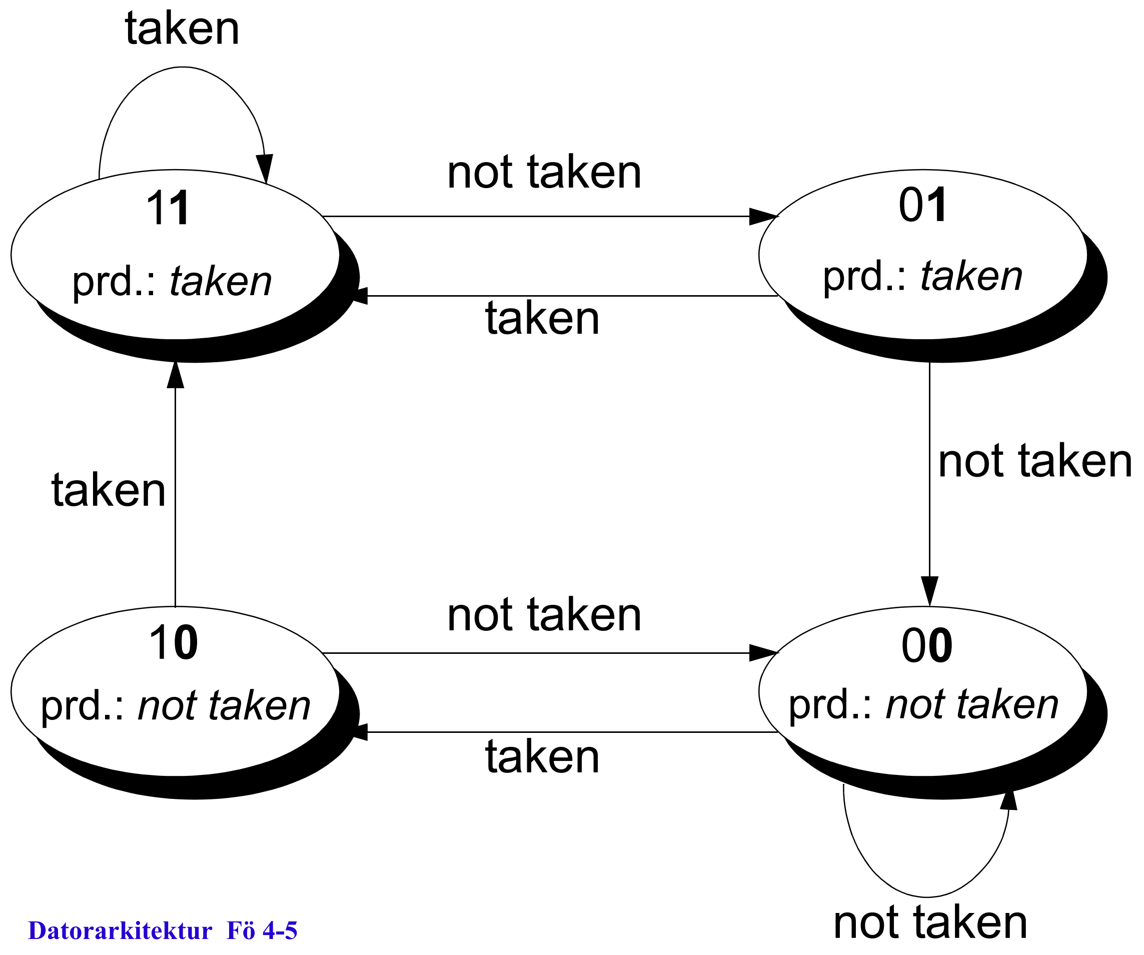
## Two-Bit Prediction Scheme

- With a two-bit scheme predictions can be made depending on the last two instances of execution.
- A typical scheme is to change the prediction only if there have been two incorrect predictions in a row.

## Dynamic Branch Prediction

**Two-Bit Prediction Scheme**

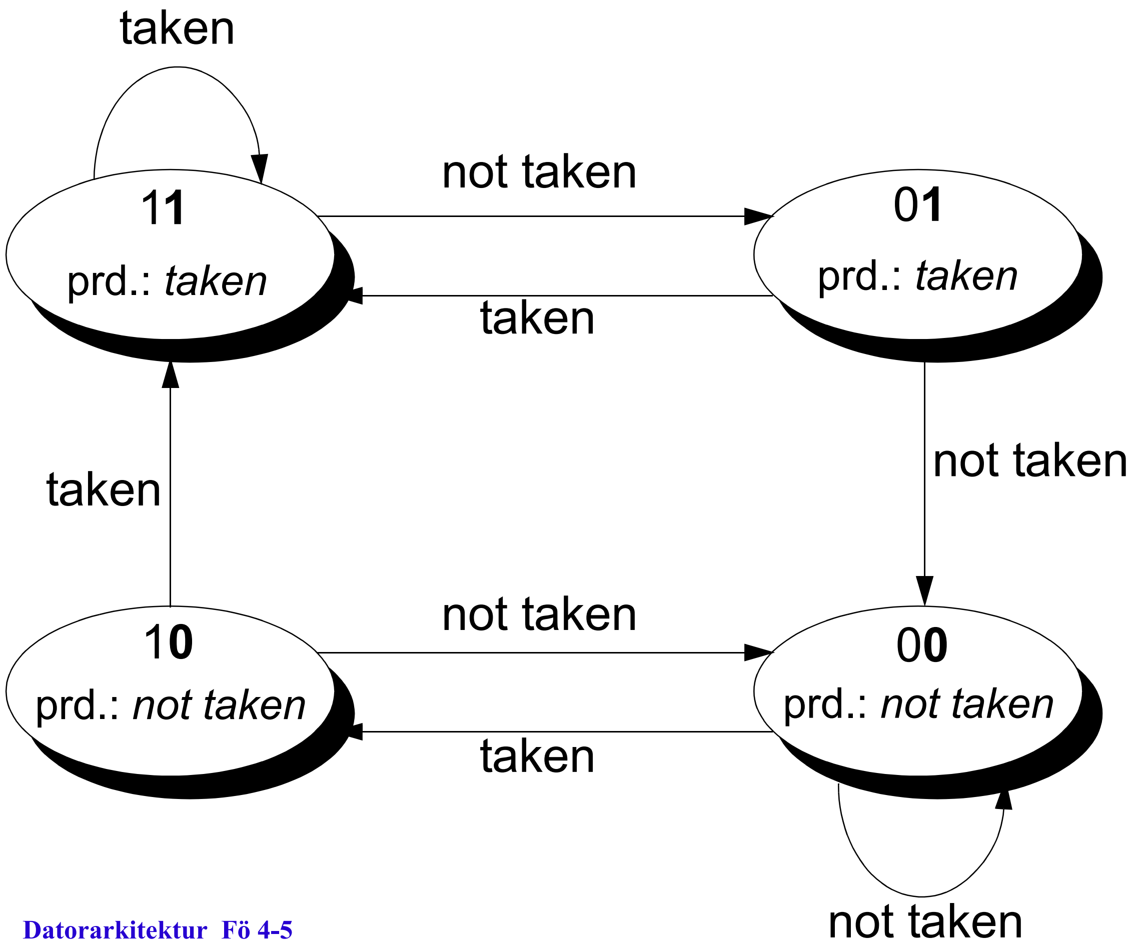
- With a two-bit scheme predictions can be made depending on the last two instances of execution.
- A typical scheme is to change the prediction only if there have been two incorrect predictions in a row.



# Dynamic Branch Prediction

## Two-Bit Prediction Scheme

- With a two-bit scheme predictions can be made depending on the last two instances of execution.
- A typical scheme is to change the prediction only if there have been two incorrect predictions in a row.

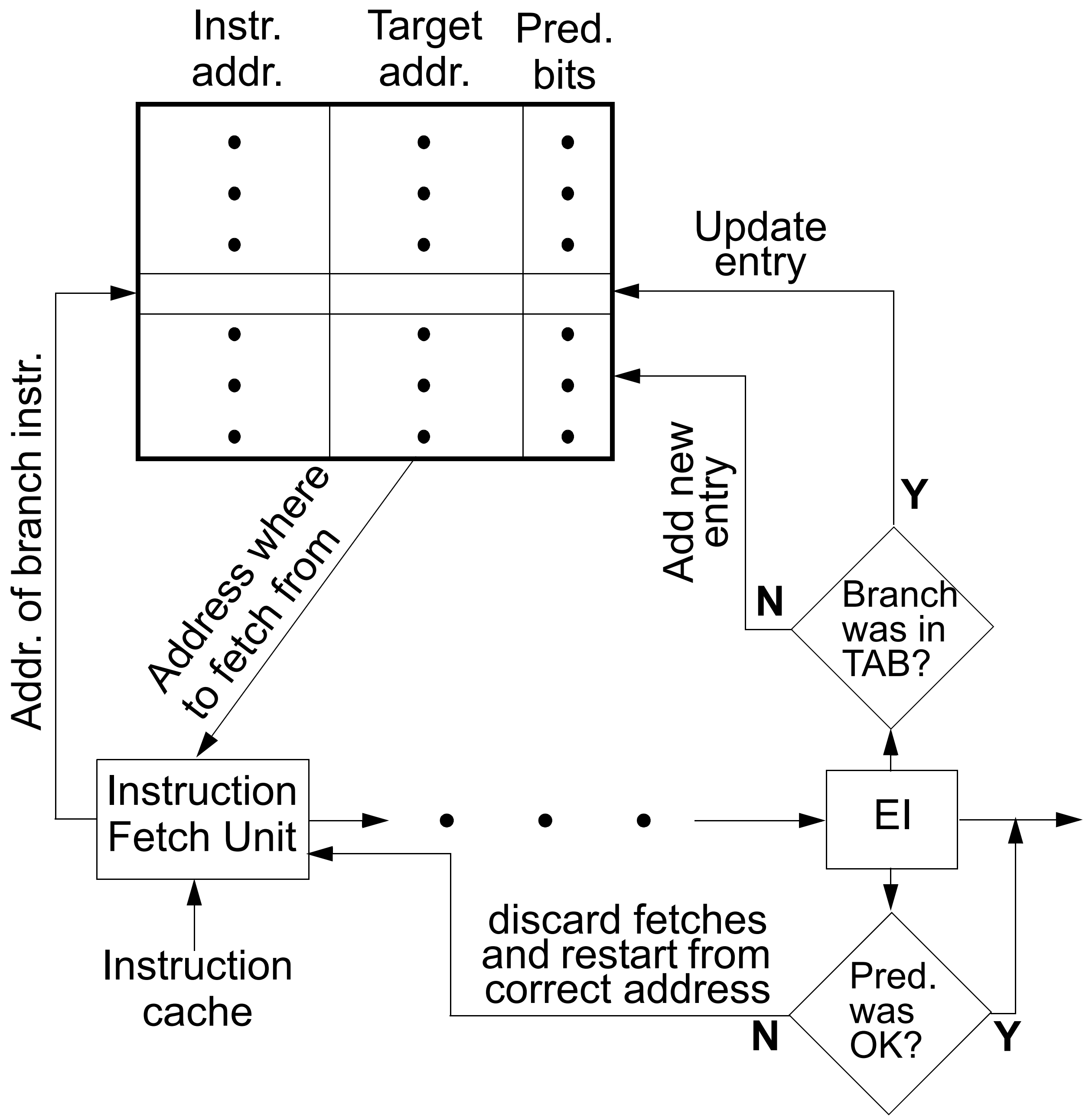


```
            - - - - - - - - - - - -
LOOP        - - - - - - - - - - - -
            - - - - - - - - - - - -
            BNZ       LOOP
            - - - - - - - - - - - -
```

**After the first execution of the loop the bits attached to BNZ will be 01; now, there will be always one false prediction for the loop, at its exit.**

## Branch History Table

**History can be used to predict the outcome of a conditional branch and to avoid recalculation of the target address. Together with the bits used for prediction, the target address is stored for later use in a *branch history table*.**



# Branch History Table

- *Address where to fetch from*: If the branch instruction is not in the table the next instruction (*address PC+1*) is to be fetched. If the branch instruction is in the table first of all a prediction based on the *prediction bits* is made. Depending on the prediction outcome the next instruction (*address PC+1*) or the instruction at the *target address* is to be fetched.
- *Update entry*: If the branch instruction has been in the table, the respective entry has to be updated to reflect the correct or incorrect prediction.
- *Add new entry*: If the branch instruction has not been in the table, it is added to the table with the corresponding information concerning branch outcome and target address. If needed one of the existing table entries is discarded. Replacement algorithms similar to those for cache memories are used.

- Using dynamic branch prediction with history tables up to 90% of predictions can be correct.
- Both Pentium and PowerPC 620, for example, use speculative execution with dynamic branch prediction based on a branch history table.

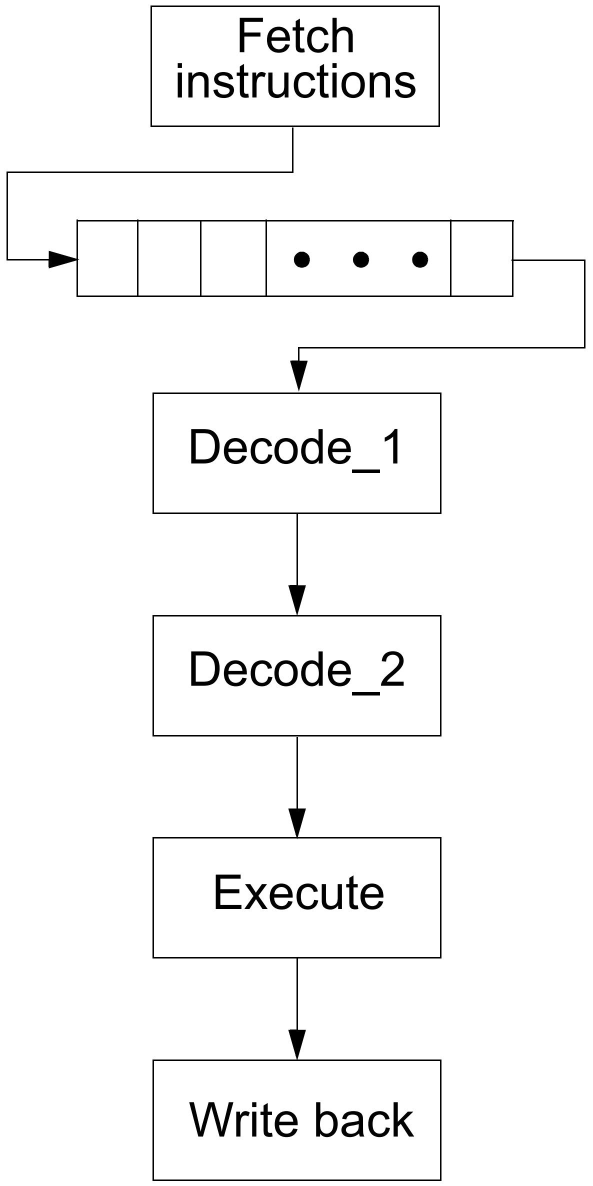
# The Intel 80486 Pipeline

- The 80486 is the last x86 processor that is not superscalar. It is a typical example of an advanced non-superscalar pipeline.

- The 80486 has a five stage pipeline.

- No branch prediction or, in fact, *always not taken*.

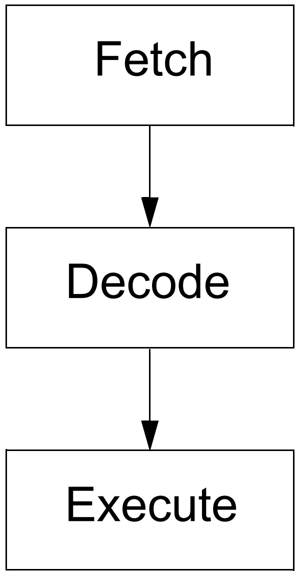
# The Intel 80486 Pipeline



- Fetch: instructions fetched from cache and placed into instruction queue (organised as two *prefetch buffers*).
  Operates independently of the other stages and tries to keep the prefetch buffers full.
- Decode_1: Takes the first 3 bytes of the instruction and decodes *opcode*, *addressing-mode*, *instruction length*; rest of the instruction is decoded by Decode_2.
- Decode_2: decodes the rest of the instruction and produces control signals; preforms address computation.
- Execute: ALU operations; cache access for operands.
- Write back: updates registers, status flags; for memory update sends values to cache and to write buffers.

# Some ARM pipelines

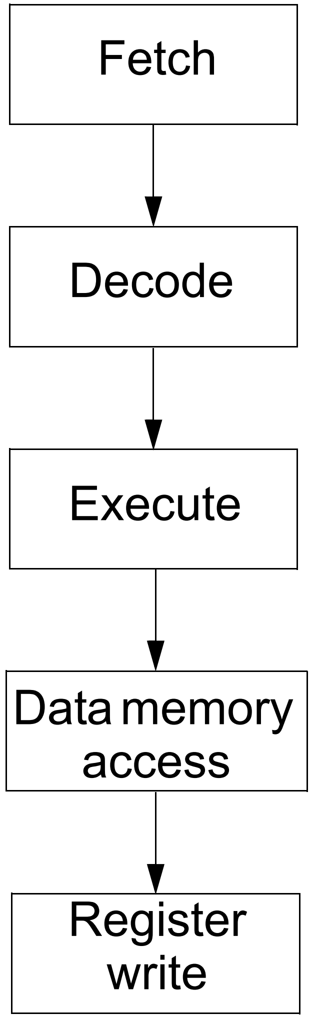
## ARM7 pipeline



- <u>Fetch</u>: instructions fetched from cache.
- <u>Decode</u>: instructions and operand registers decoded.
- <u>Execute</u>: registers read; shift and ALU operations; results or loaded data from memory written back to register.

# Some ARM pipelines

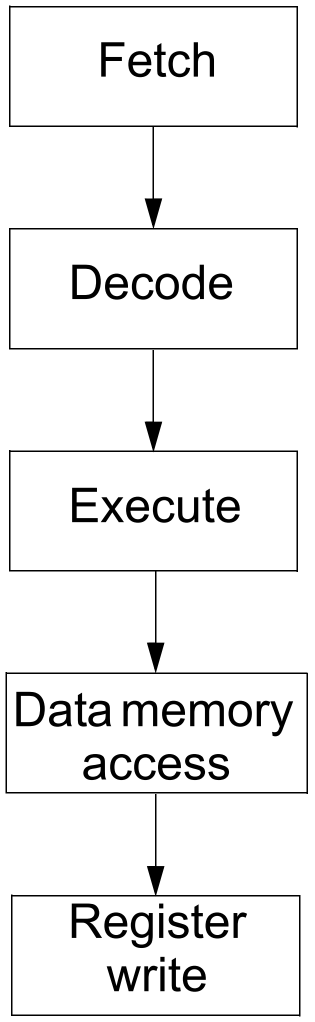
## ARM9 pipeline



- Fetch: instructions fetched from I-cache.
- Decode: instructions and operand registers decoded; registers read.
- Execute: shift and ALU operations (if load/store, then memory address computed).
- Data memory access: fetch/store data from/to D-cache (if no memory access, the ALU result is buffered for one cycle; this is lost time!).
- Register write: results or loaded data written back to register.

# Some ARM pipelines

## ARM9 pipeline



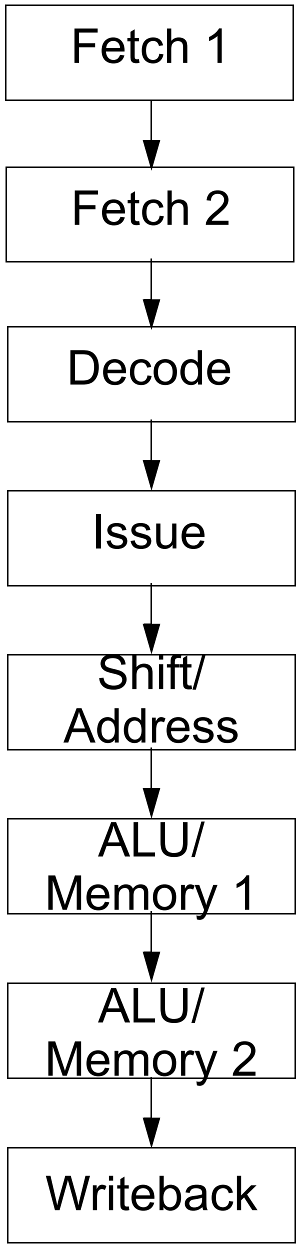
- Fetch: instructions fetched from I-cache.
- Decode: instructions and operand registers decoded; registers read.
- Execute: shift and ALU operations (if load/store, then memory address computed).
- Data memory access: fetch/store data from/to D-cache (if no memory access, the ALU result is buffered for one cycle; this is lost time!).
- Register write: results or loaded data written back to register.

The performance of the ARM9 is significantly superior to the ARM7:

- Higher clock speed due to larger number of pipeline stages.
- More even distribution of tasks among pipeline stages; tasks have been moved away from the execute stage.

# Some ARM pipelines

## ARM11 pipeline

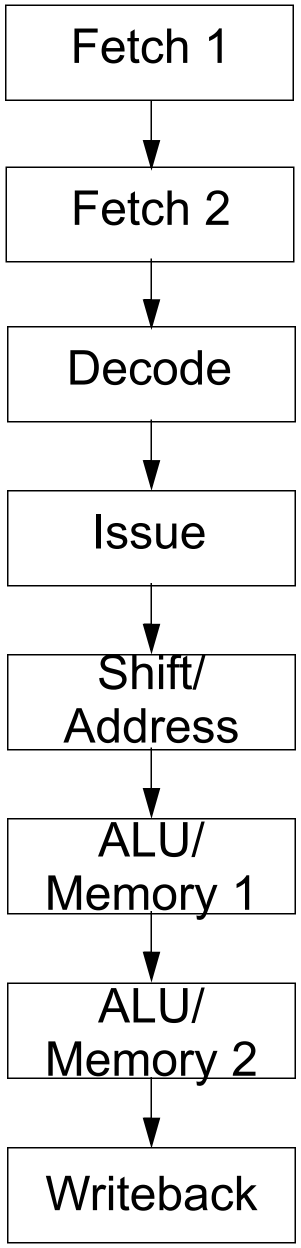


**The performance of ARM11 is further enhanced by:**

- **Higher clock speed due to larger number of pipeline stages; more even distribution of tasks among pipeline stages.**
- **Branch prediction:**
  - **Dynamic two bits prediction based on a 64 entry branch history table (branch target address cache - BTAC).**
  - **If the instruction is not in the BTAC, static prediction is done: *taken* if backward, *not taken* if forward.**

# Some ARM pipelines

## ARM11 pipeline

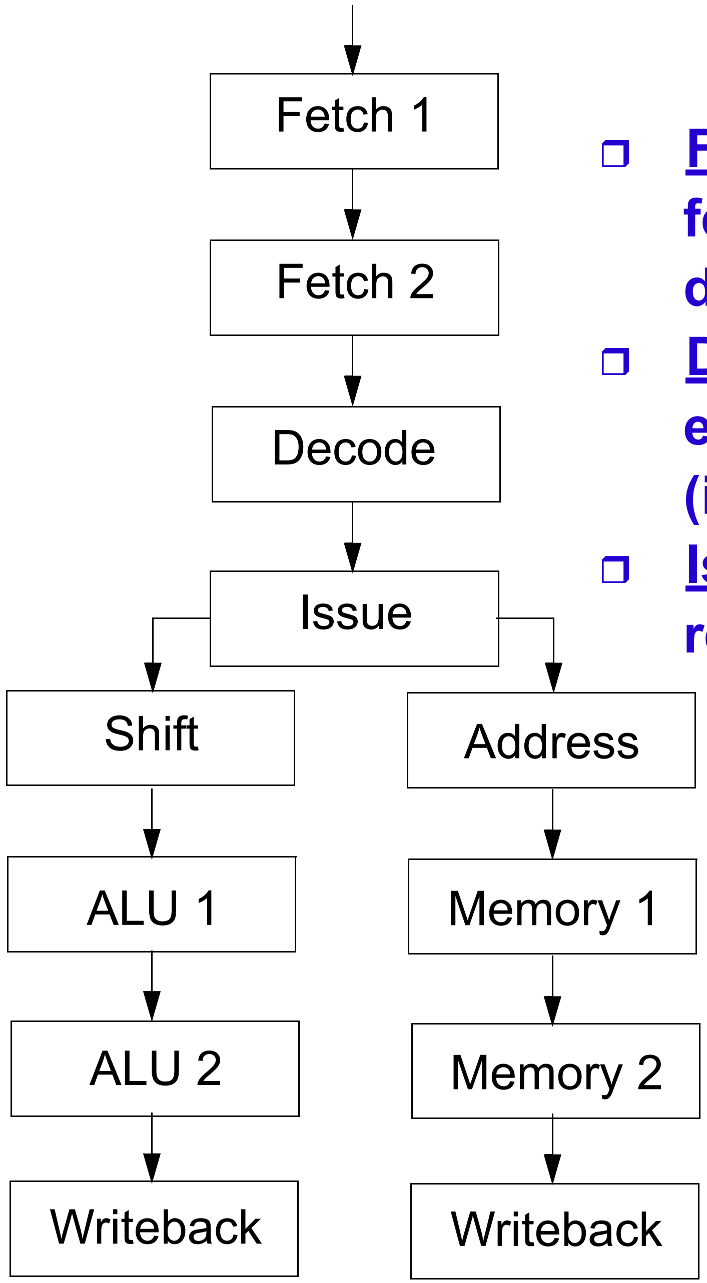


The performance of ARM11 is further enhanced by:

- Higher clock speed due to larger number of pipeline stages; more even distribution of tasks among pipeline stages.
- Branch prediction:
  - Dynamic two bits prediction based on a 64 entry branch history table (branch target address cache - BTAC).
  - If the instruction is not in the BTAC, static prediction is done: *taken* if backward, *not taken* if forward.
- Decoupling of the load/store pipeline from the ALU&MAC (multiply-accumulate) pipeline: ALU operations can work for one instruction while load/store operations complete for another one.

# Some ARM pipelines

## ARM11 pipeline



- **<u>Fetch 1, 2</u>: instructions fetched from I-cache; dynamic branch prediction.**
- **<u>Decode</u>: instructions decoded; static branch prediction (if needed).**
- **<u>Issue</u>: instruction issued; registers read.**
- **<u>Shift</u>: register shift/ rotate.**
- **<u>ALU 1,2</u>: ALU/MAC operations.**
- **<u>Writeback</u>: results written to register.**
- **<u>Address</u>: address calculation.**
- **<u>Memory</u> 1,2: data memory access.**
- **<u>Writeback</u>: write loaded data to reg.; commit store.**