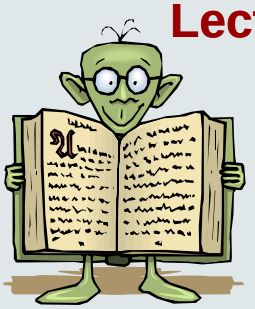
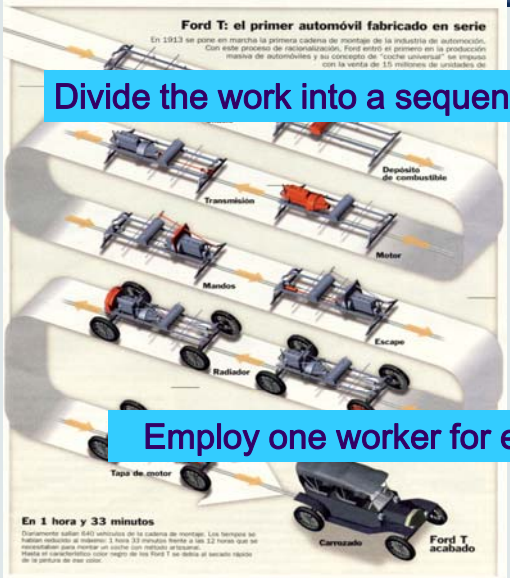
# Lecture 3: Instruction Pipelining



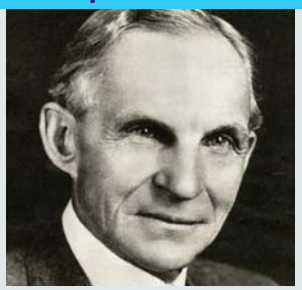
- Basic concepts
- Pipeline hazards
- Branch handling
- Branch prediction



# What is a Pipeline?



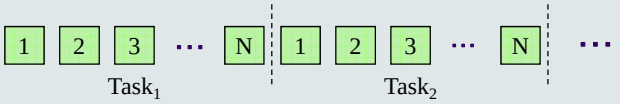

Divide the work into a sequence of simpler tasks.



Employ one worker for each simpler task.

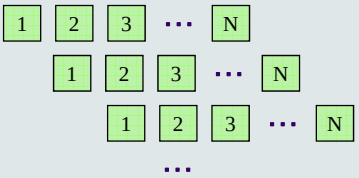
# Basic Concepts

- Sequential execution of an N-stage task:




  - Production time: N time units.
  - Resource needed: one general-purpose machine.
  - Productivity: one product per N time units.

- Pipelined execution of an N-stage task:



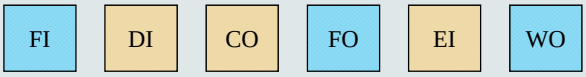

  - Production time: N time units.
  - Resource needed: N special-purpose machines.
  - Productivity: about one product per time unit.



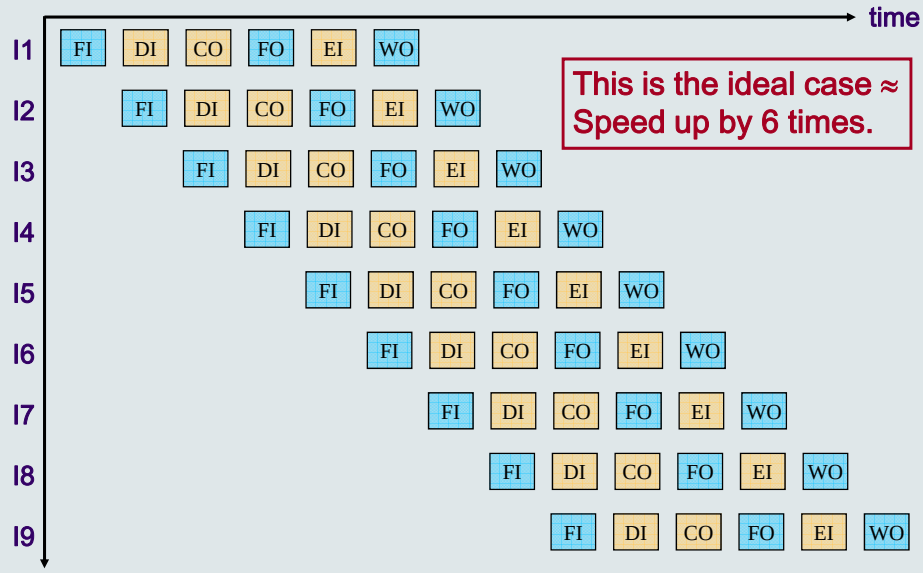
# Instruction Execution Stages

A typical instruction execution sequence:

1. Fetch Instruction (FI): Fetch the instruction.
2. Decode Instruction (DI): Determine the op-code and the operand specifiers.
3. Calculate Operands (CO): Calculate the effective addresses.
4. Fetch Operands (FO): Fetch the operands.
5. Execute Instruction (EI): perform the operation.
6. Write Operand (WO): store the result in memory.



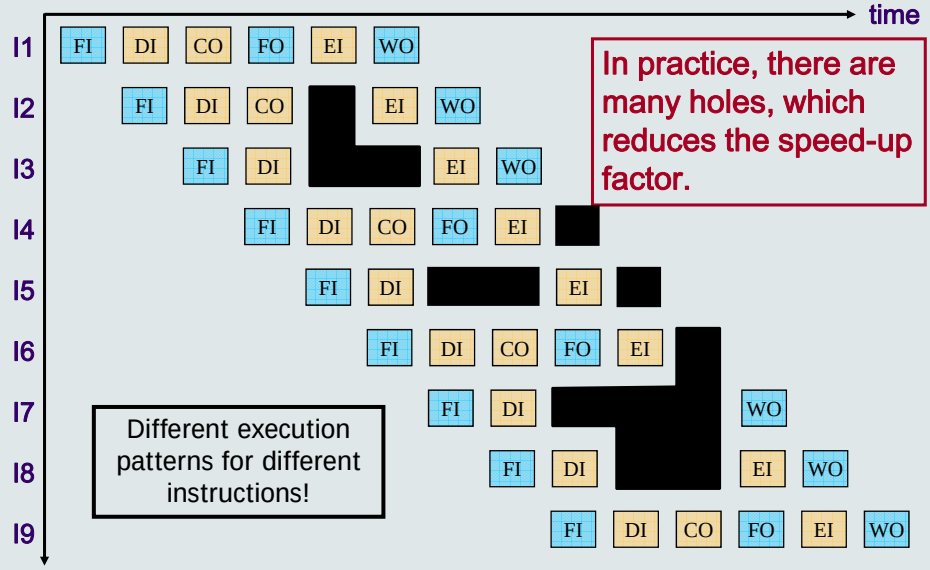
# Instruction Pipelining




This is the ideal case ≈ Speed up by 6 times.



# Typical Instruction Pipelining




In practice, there are many holes, which reduces the speed-up factor.

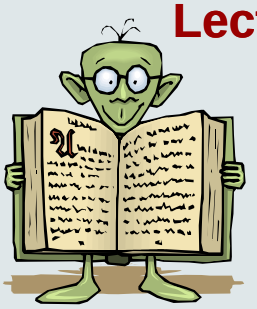
Different execution patterns for different instructions!

# Number of Pipeline Stages

- In general, a larger number of stages gives better performance.
- However:
  - A larger number of stages increases the overhead in moving information between stages and synchronization between stages.
  - The complexity of the CPU grows with the number of stages.
  - It is difficult to keep a large pipeline at maximum rate because of pipeline hazards.
- Intel 80486 and Pentium:
  - Five-stage pipeline for integer instructions.
  - Eight-stage pipeline for FP (floating points) instructions.
- IBM PowerPC:
  - Four-stage pipeline for integer instructions.
  - Six-stage pipeline for FP instructions.



# Lecture 3: Instruction pipelining



- **Basic concepts**
- **Pipeline hazards**
- **Branch handling**
- **Branch prediction**

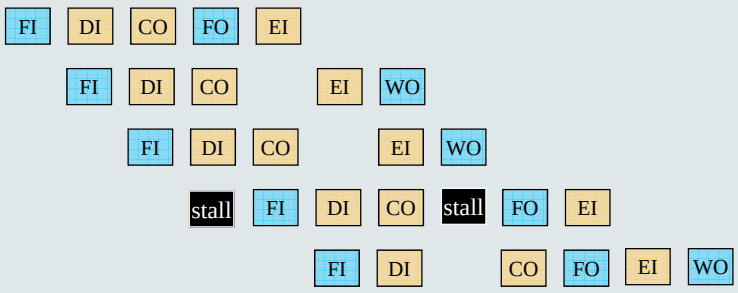
# Pipeline Hazards (Conflicts)

- They 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;
  - No new instructions are fetched during the stall;
  - Instructions earlier than the stalled one continue as usual.
- Types of hazards:
  - Structural hazards
  - Data hazards
  - Control hazards



# Structural (Resource) Hazards

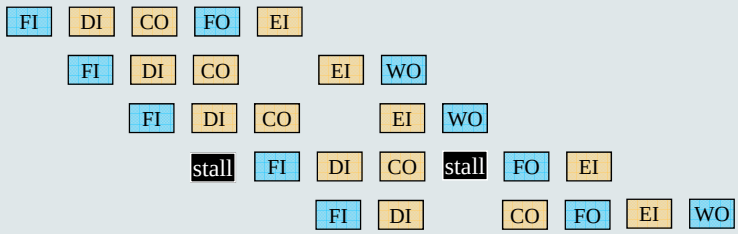
- Hardware conflicts — caused by the use of the same hardware resource at the same time (e.g., memory conflicts).




- Penalty: 1 cycle (NOTE: the performance lost is multiplied by the number of stages).

# Structural Hazard Solutions

- In general, the hardware resources in conflict are duplicated in order to avoid structural hazards.
- Functional units (ALU, FP unit) can also be pipelined themselves to support several instructions at the same time.
- Memory conflicts can be solved by:
  - having two separate caches, one for instructions and the other for operands (Harvard architecture);





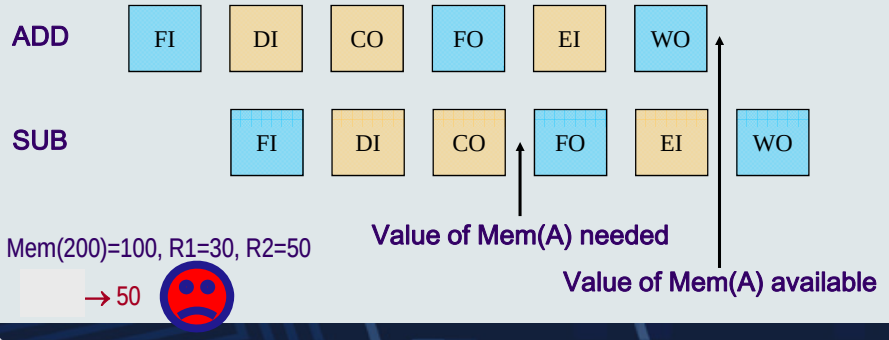

# Structural Hazard Solutions

- In general, the hardware resources in conflict are duplicated in order to avoid structural hazards.
- Functional units (ALU, FP unit) can also be pipelined themselves to support several instructions at the same time.
- Memory conflicts can be solved by
  - having two separate caches, one for instructions and the other for operands (Harvard architecture);
  - Using multiple banks of the main memory; or
  - keeping as many intermediate results as possible in the registers (!).

# Data Hazards

- Caused by reversing the order of data-dependent operations due to the pipeline (e.g., WRITE/READ conflicts).

ADD A, R1; Mem(A) ← Mem(A) + R1;
SUB A, R2; Mem(A) ← Mem(A) - R2;



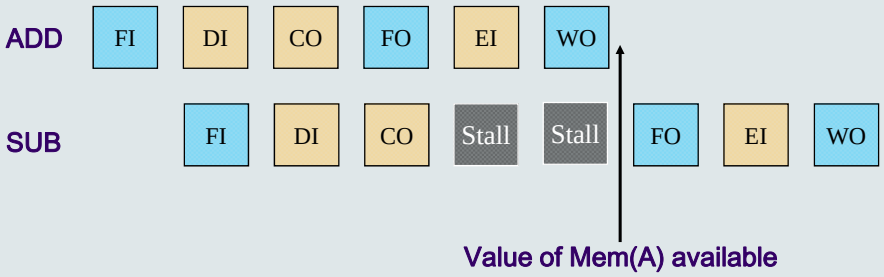

Mem(200)=100, R1=30, R2=50

→ 50



# Data Hazard Penalty

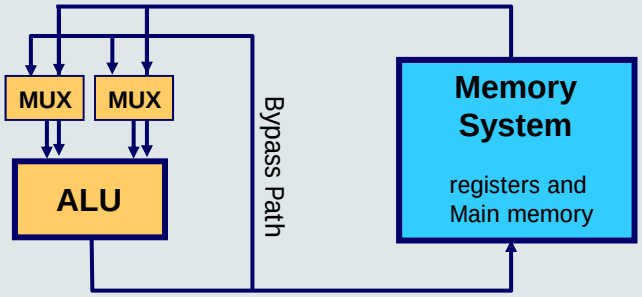
ADD A, R1; Mem(A) ← Mem(A) + R1;
SUB A, R2; Mem(A) ← Mem(A) - R2;




- Penalty: 2 cycles.

# Data Hazard Solutions

- The penalty due to data hazards can be reduced by a technique called forwarding (bypassing).

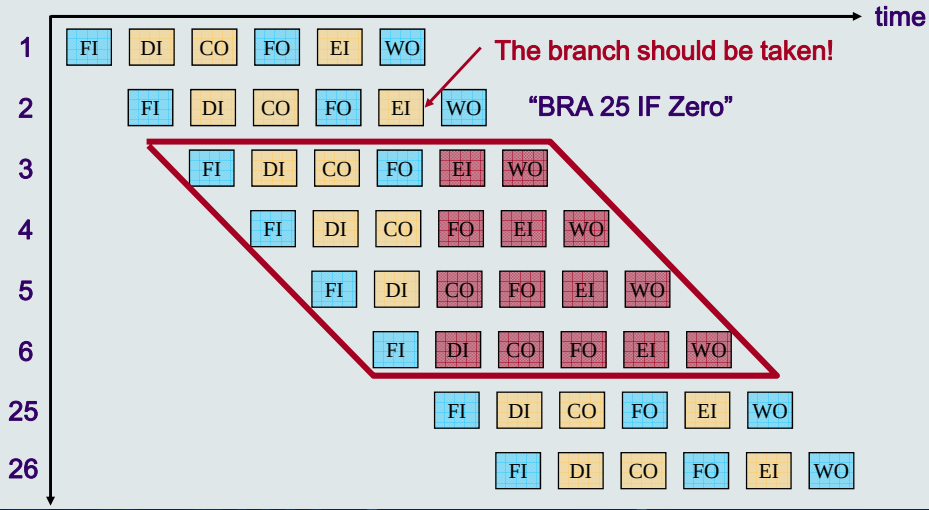



- The ALU result is fed back to the ALU input.
  - If it detects that the value needed for an operation is the one produced by the previous operation (but which has not yet been written back), it selects the forwarded result, instead of the value from the memory system.

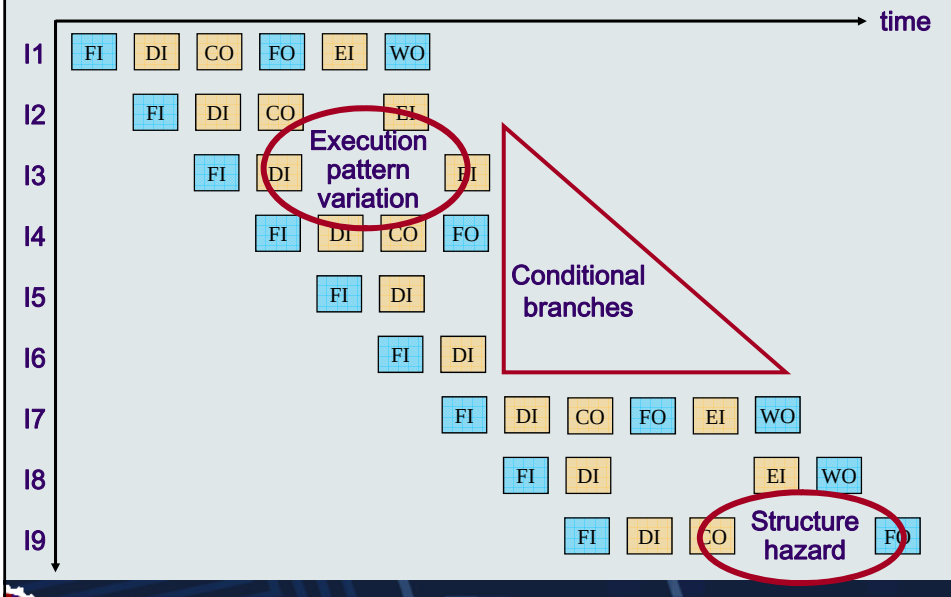


# Control Hazards

- Caused by branch instructions, which change the instruction execution order.

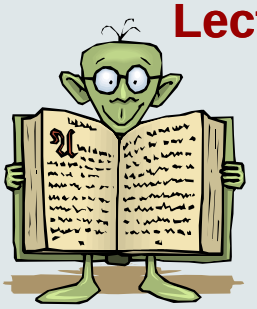


# Instruction Pipelining in Practice





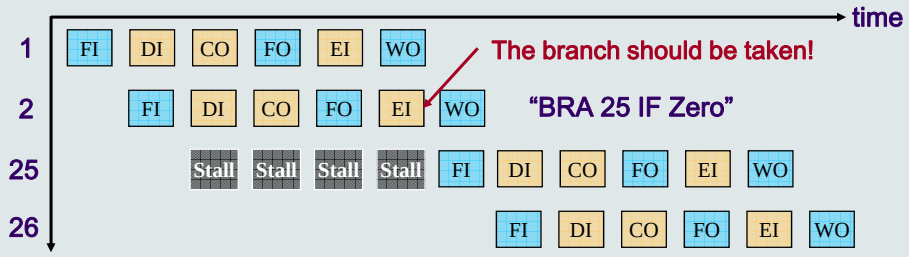

# Lecture 3: Instruction pipelining



- Basic concepts
- Pipeline hazards
- Branch handling
- Branch prediction

# Branch Handling (1)

- Stop the pipeline until the branch instruction reaches the last stage.

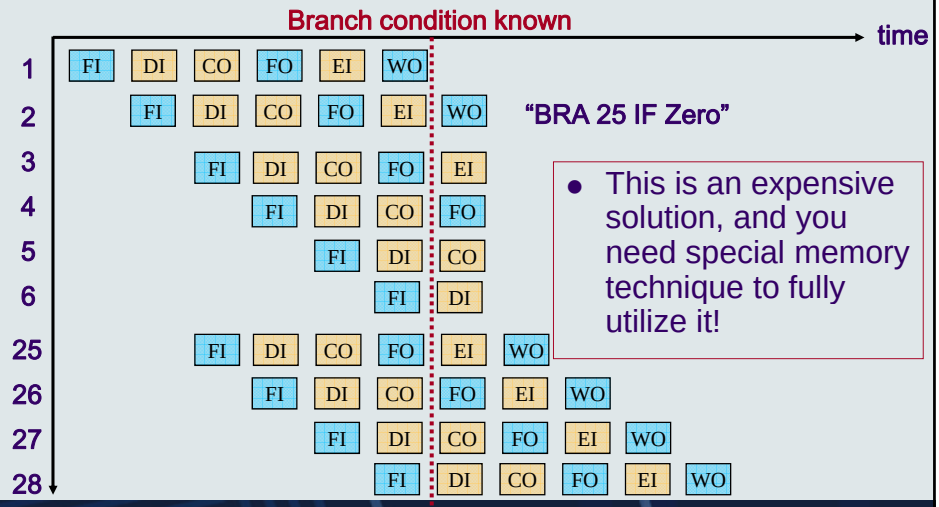



- Large lost of performance, since 20% - 35% of the instructions executed are branches.



# Branch Handling (2)

- Multiple streams — implement hardware resources to deal with different branch alternatives.




- This is an expensive solution, and you need special memory technique to fully utilize it!

# Branch Handling (3)

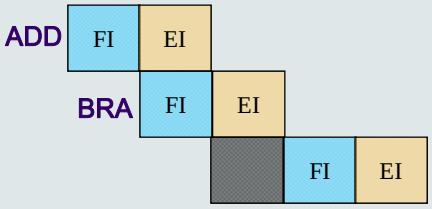
- Pre-fetch branch target — when a conditional branch is recognized, the following instruction is fetched, and the branch target is also pre-fetched.
- Loop buffer — use a small, very high-speed memory to keep the *n* most recently fetched instructions in sequence. If a branch is to be taken, the buffer is first checked to see if the branch target is in it.
  - Special cache for branch target instructions.
- Delayed branch — re-arrange the instructions so that branching occur later than originally specified.



# Delayed Branch Example

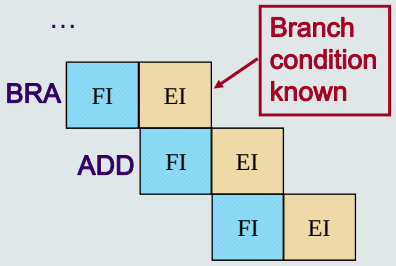
**Original inst. sequence:**

ADD X;
BRA L;
…



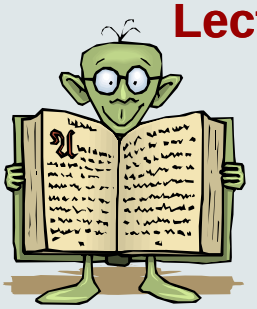

**Delayed branch:**

BRA L;
ADD X;
…




- **The compiler or the programmer** has to find an instruction which can be moved from its original place to the branch delay slot (it will be executed regardless of the branch outcome).
  - 60% to 85% success rate.

# Lecture 3: Instruction pipelining



- Basic concepts
- Pipeline hazards
- Branch handling
- Branch prediction

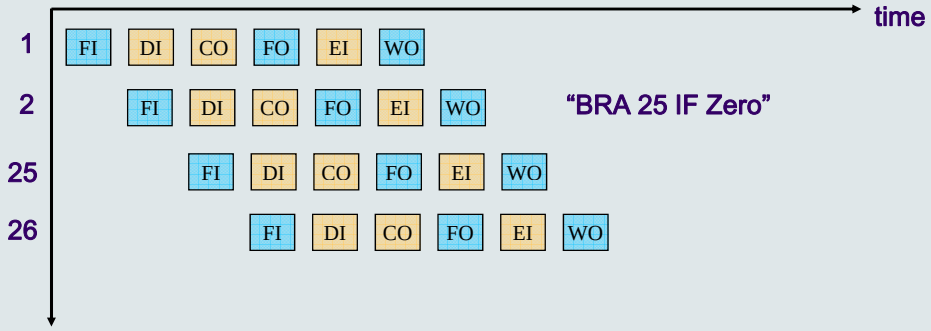


# Branch Prediction

- When a branch is encountered, a prediction is made and the predicted path is followed.
- The instructions on the predicted path are fetched.
- The fetched instruction can also be executed — called **Speculative Execution**.
  - The results produced of these executions should be marked as tentative.
- When the branch outcome is decided, if the prediction is correct, the special tags on tentative results are removed.
- If not, the tentative results are removed. And the execution goes to the other path.
- Branch prediction can base on static information or dynamic information.

# Static Branch Prediction

- Predict always taken
  - Assume that jump will happen.
  - Always fetch target instruction.






# Static Branch Prediction (Cont'd)

```
sum = 0;        //Java code
for (i = 0; i < 1000; i++)
sum += a[i];
```

```
     MOVE #0, R0    -- sum
     MOVE #100, R1 -- index
L1:  ADD  R0, (R1)
     ADD  R1, #1
     COMP R1, #'1100'
     BNZ  L1
     MOVE R0, sum
```

The prediction will be correct 99.9% of the time with this example !

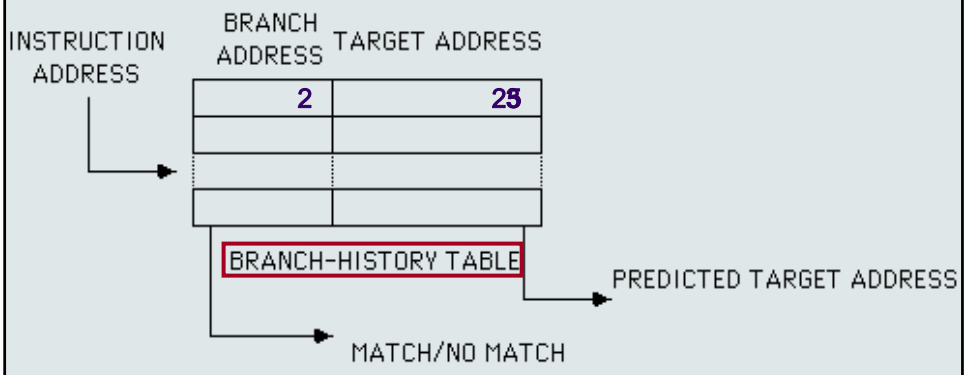
# Static Branch Prediction (Cont'd)

- Predict never taken
  - Assume that jump will not happen.
  - Always fetch next instruction.
- Predict by Operation Codes
  - Some instructions are more likely to result in a jump than others.
    - BNZ (Branch if the result is Not Zero)
    - BEZ (Branch if the result equals Zero)
  - Can get up to 75% success.



# Dynamic Branch Prediction

- Based on branch history:
  - Store information regarding branches in a branch-history table so as to more accurately predict the branch outcome.
  - Assume that the branch will do what it did last time.



# Summary

- Instruction execution can be substantially accelerated by instruction pipelining.
- A pipeline is organized as a succession of *N* stages. Ideally *N* instructions can be active inside a pipeline.
- Keeping a pipeline at its maximal rate is, however, prevented by pipeline hazards.
  - Structural hazards are due to resource conflicts.
  - Data hazards are caused by data dependencies between instructions.
  - Control hazards are produced as consequence of branch instructions.
- Branch instructions can dramatically affect pipeline performance. It is very important to reduce penalties produced by branches.