Lecture-6

Chapter-14.4

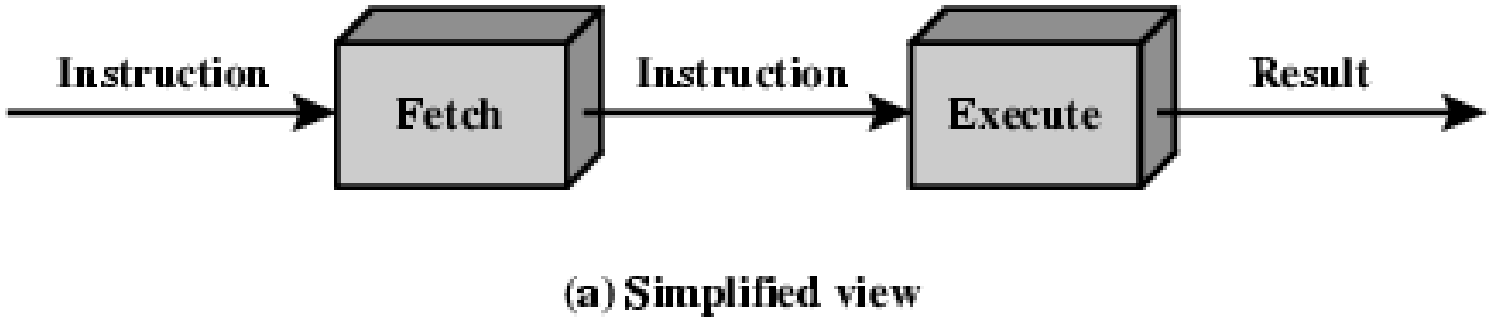
Computer Organization and Architecture Designing - William Stallings

Instruction Pipelining

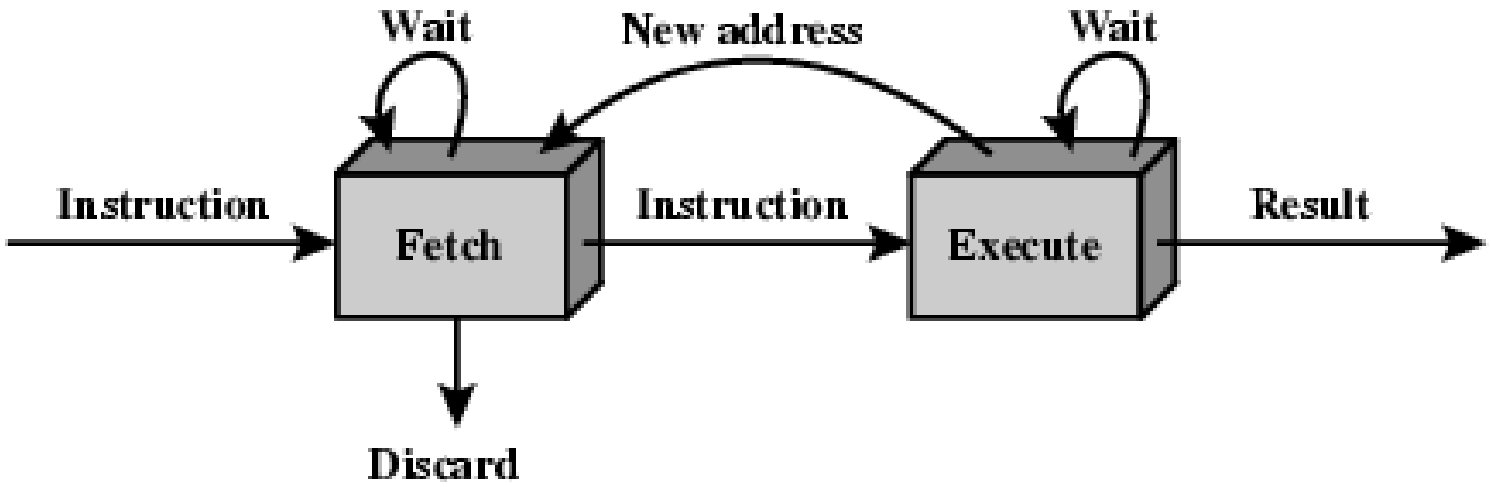
Stages of Pipelining

- Fetch Instruction (FI)
- Decode Instruction (DI)
- Calculate Operands (CO)
- Fetch Operands (FO)
- Execute Instructions (EI)
- Write Operands (WO)

Two Stage Instruction Pipeline

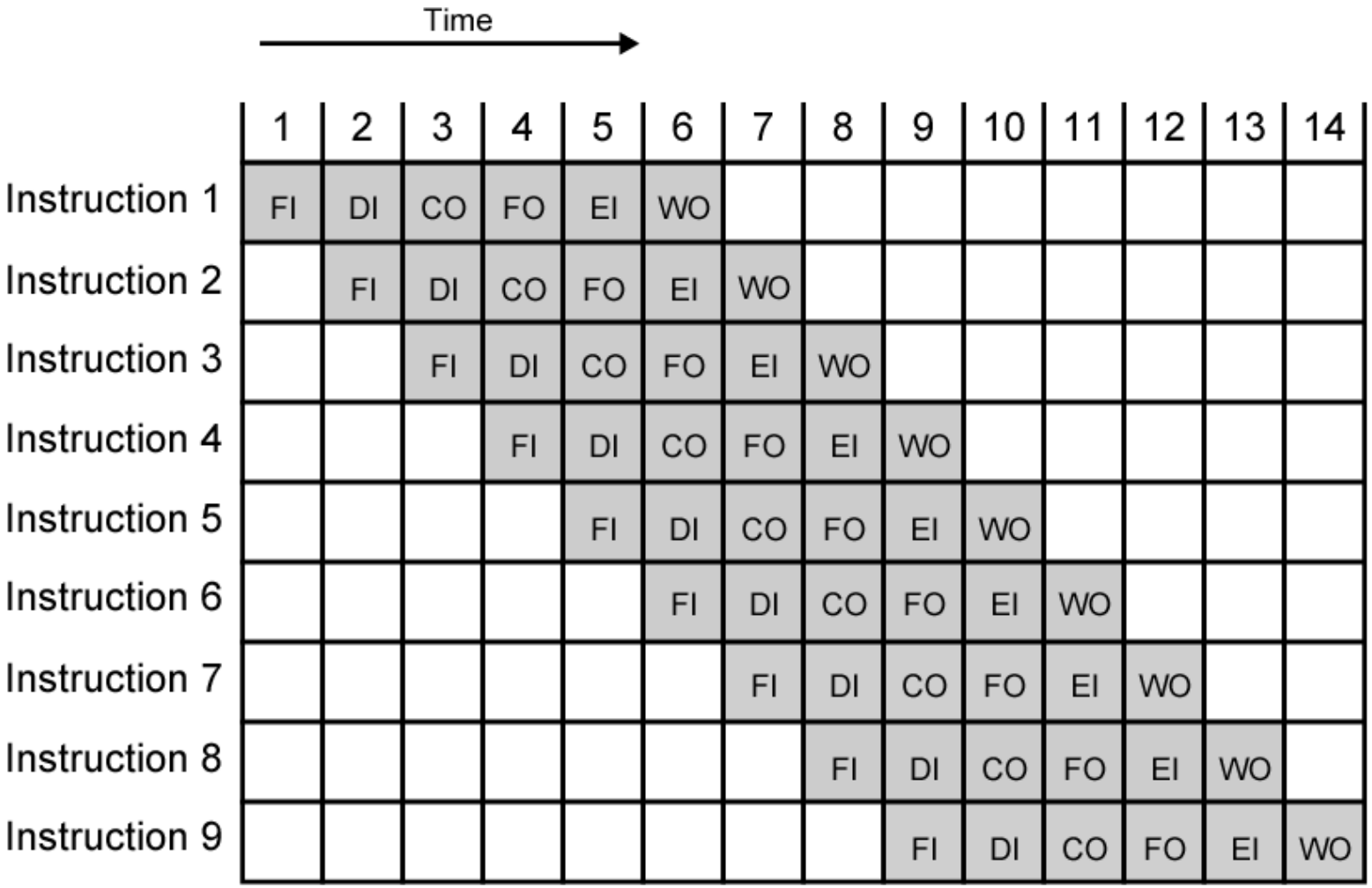


(a) Simplified view

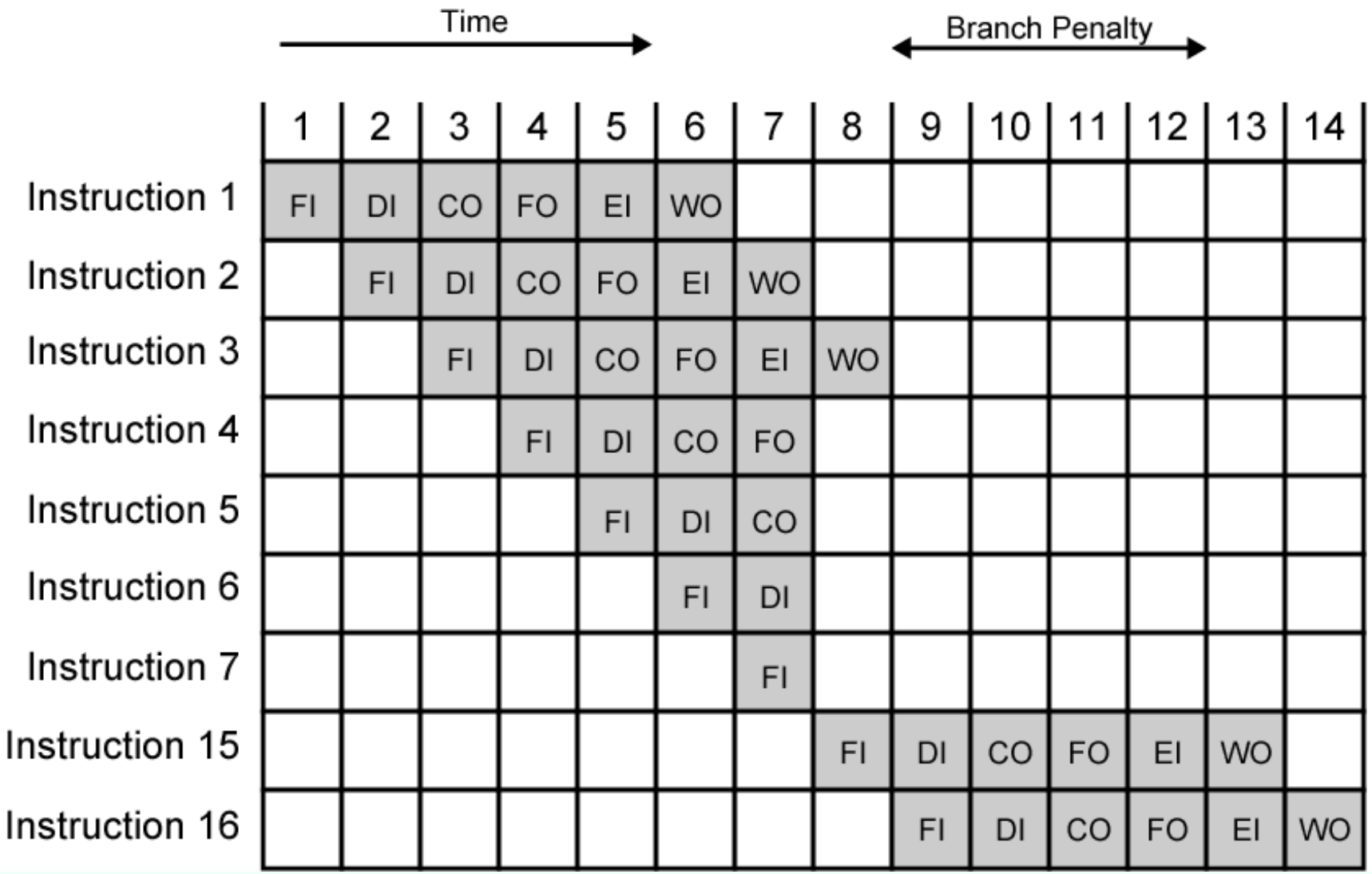


(b) Expanded view

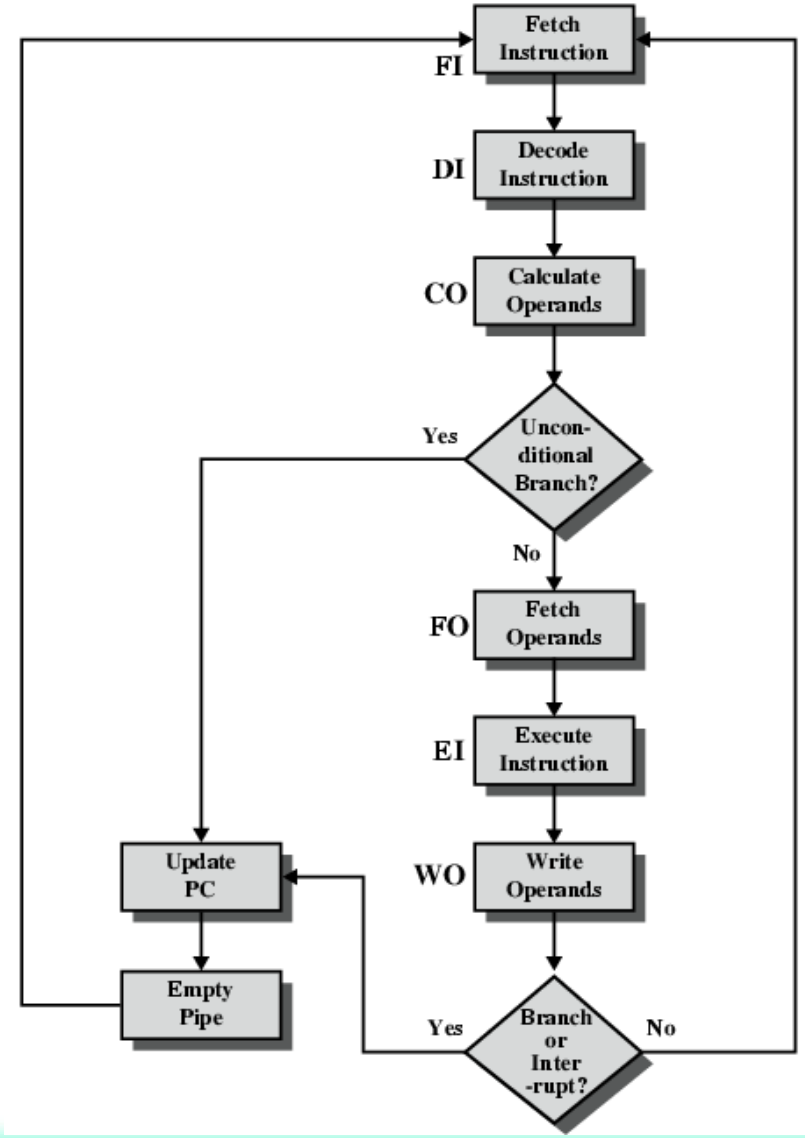
Timing Diagram for Instruction Pipeline Operation



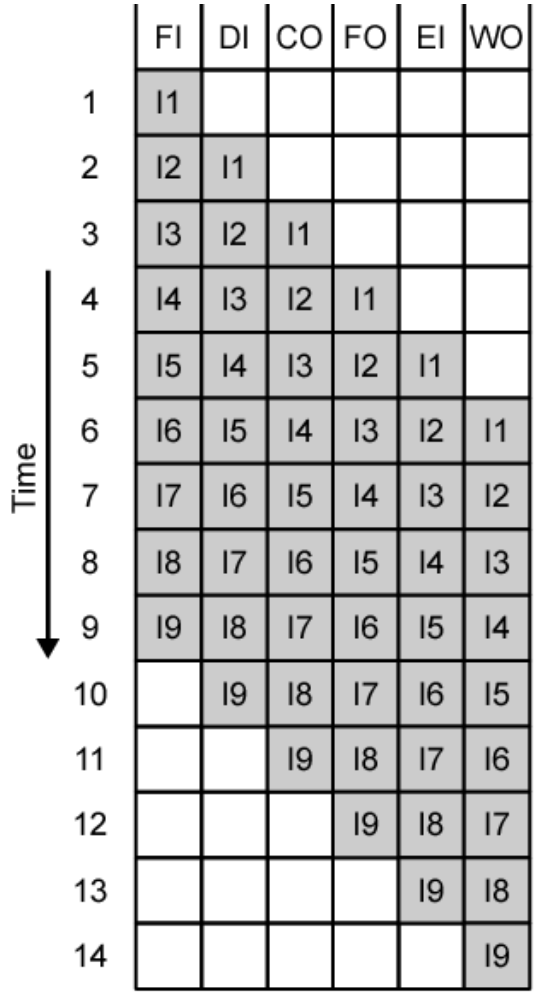
The Effect of a Conditional Branch on Instruction Pipeline Operation



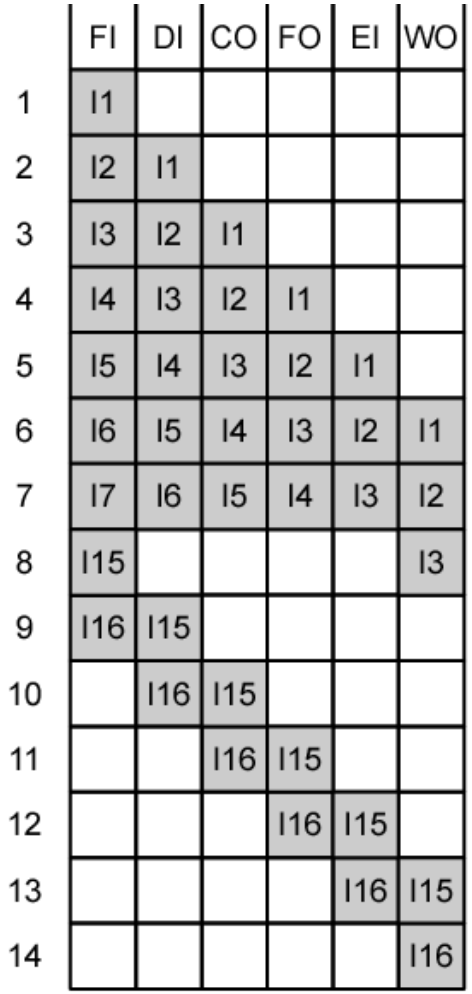
Six Stage Instruction Pipeline



Alternative Pipeline Depiction



(a) No branches



(b) With conditional branch

Pipeline Hazards

- Pipeline, or some portion of pipeline, must stall
- Also called *pipeline bubble*
- Types of hazards
  - Resource
  - Data
  - Control

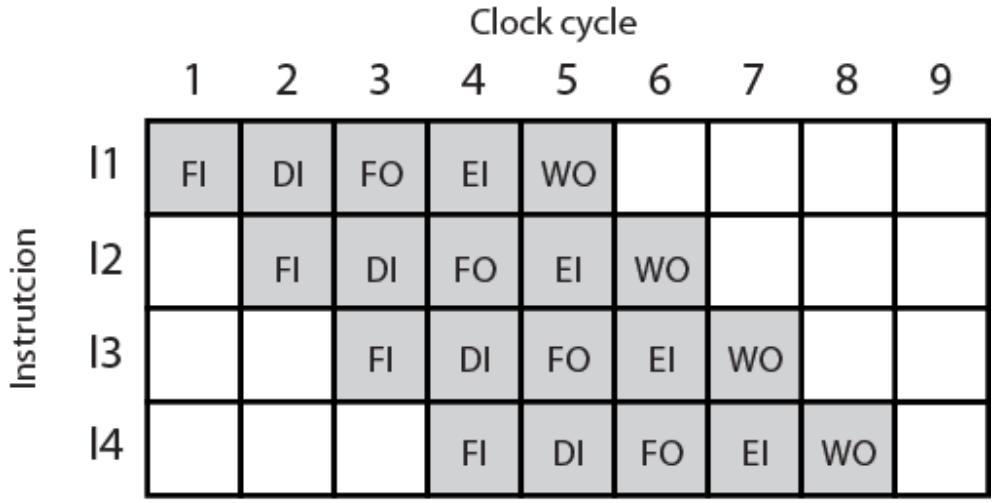
Resource Hazards

- Two (or more) instructions in pipeline need same resource
- Executed in serial rather than parallel for part of pipeline
- Also called *structural hazard*
- E.g. Assume simplified five-stage pipeline
  - Each stage takes one clock cycle
- Ideal case is new instruction enters pipeline each clock cycle
- Assume main memory has single port
- Assume instruction fetches and data reads and writes performed one at a time
- Ignore the cache
- Operand read or write cannot be performed in parallel with instruction fetch
- Fetch instruction stage must idle for one cycle fetching I3

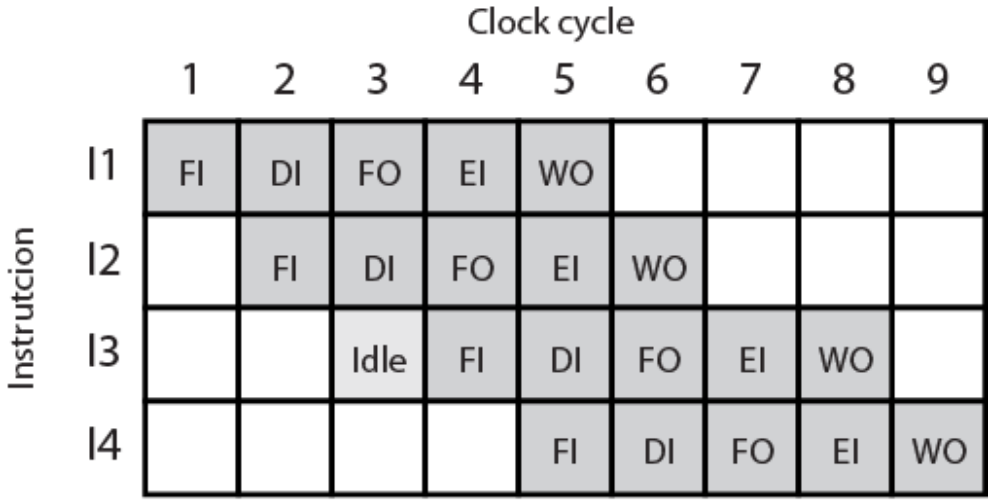
- E.g. multiple instructions ready to enter execute instruction phase
- Single ALU

- One solution: increase available resources
  - Multiple main memory ports
  - Multiple ALUs

Resource Hazard Diagram



(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

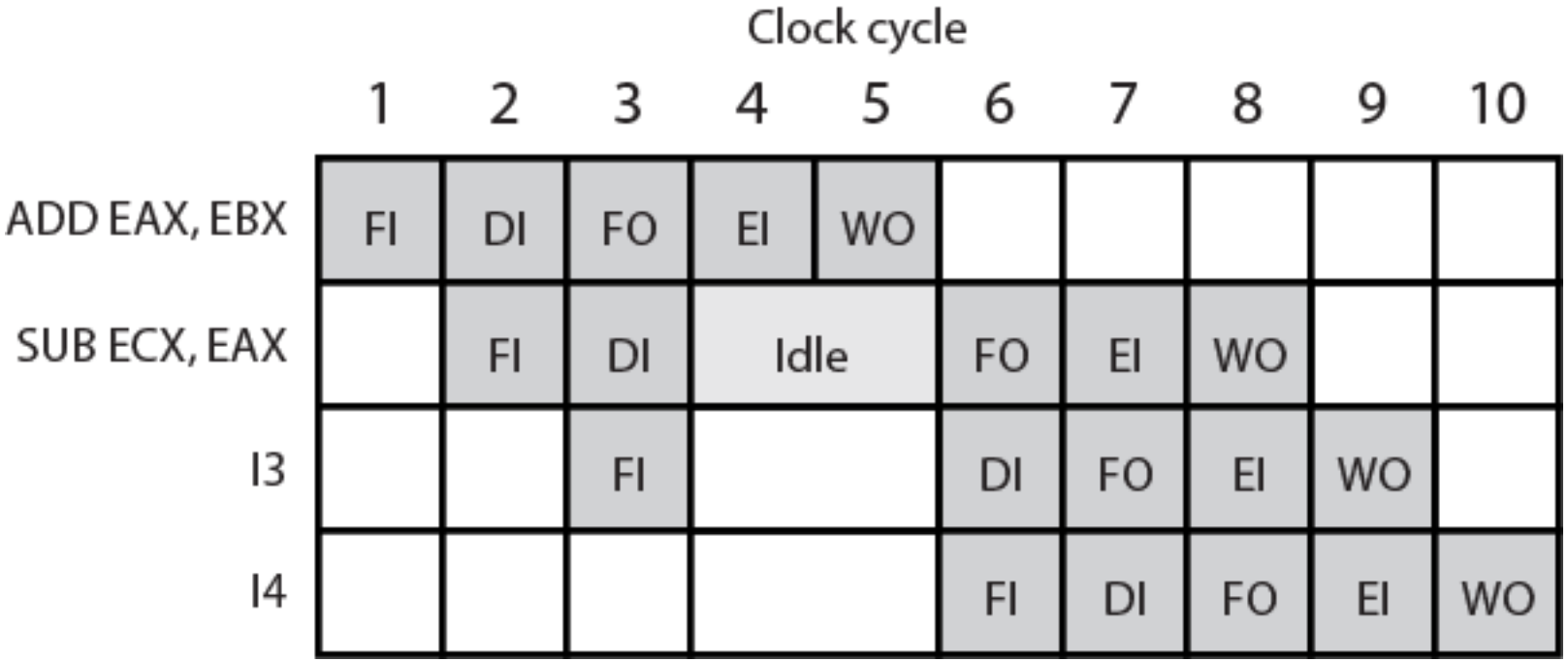
Data Hazards

- Conflict in access of an operand location
- Two instructions to be executed in sequence
- Both access a particular memory or register operand
- If in strict sequence, no problem occurs
- If in a pipeline, operand value could be updated so as to produce different result from strict sequential exe cution
- E.g. x86 machine instruction sequence:

- ADD EAX, EBX /* EAX = EAX + EBX
- SUB ECX, EAX /* ECX = ECX – EAX

- ADD instruction does not update EAX until end of stage 5, at clock cycle 5
- SUB instruction needs value at beginning of its stage 2, at clock cycle 4
- Pipeline must stall for two clocks cycles
- Without special hardware and specific avoidance algorithms, results in inefficient pipeline usage

Data Hazard Diagram



Types of Data Hazard

- Read after write (RAW), or true dependency
  - An instruction modifies a register or memory location
  - Succeeding instruction reads data in that location
  - Hazard if read takes place before write complete
- Write after read (WAR), or antidependency
  - An instruction reads a register or memory location
  - Succeeding instruction writes to location
  - Hazard if write completes before read takes place
- Write after write (WAW), or output dependency
  - Two instructions both write to same location
  - Hazard if writes take place in reverse of order intended sequence
- Previous example is RAW hazard

Write After Read (WAR)

- (i2 tries to write a destination before it is read by i1) A write after read (WAR) data haza rd represents a problem with concurrent execution.
- **Example**
- For example:
- i1. R4 <- R1 + R3
  i2. R3 <- R1 + R2
- If we are in a situation that there is a chance that i2 may be completed before i1 (i.e. wit h concurrent execution) we must ensure that we do not store the result of R3 before i1 h as had a chance to fetch the operands.

Write After Write (WAW)

- (i2 tries to write an operand before it is written by i1) A write after write (WAW) data h azard may occur in a concurrent execution environment.
- **Example**
- For example:
- i1. R2 <- R4 + R7
  i2. R2 <- R1 + R2
- We must delay the WB (Write Back) of i2 until the execution of i1.

Control Hazard

- Also known as *branch hazard*
- Pipeline makes wrong decision on branch prediction
- Brings instructions into pipeline that must subsequently be discarded
- Dealing with Branches
  - Multiple Streams
  - Prefetch Branch Target
  - Loop buffer
  - Branch prediction
  - Delayed branching

Multiple Streams

- Have two pipelines
- Prefetch each branch into a separate pipeline
- Use appropriate pipeline

- Leads to bus & register contention
- Multiple branches lead to further pipelines being needed

Prefetch Branch Target

- Target of branch is prefetched in addition to instructions following branch
- Keep target until branch is executed
- Used by IBM 360/91

Loop Buffer

- Very fast memory
- Maintained by fetch stage of pipeline
- Check buffer before fetching from memory
- Very good for small loops or jumps
- c.f. cache
- Used by CRAY-1

Branch Prediction

Various techniques can be used to predict whether a branch will be taken o r not.

- Predict never taken
  - Assume that jump will not happen
  - Always fetch next instruction
  - 68020 & VAX 11/780
  - VAX will not prefetch after branch if a page fault would result (O/S v CPU design)
- Predict always taken
  - Assume that jump will happen
  - Always fetch target instruction

Branch Prediction

- Predict by Opcode
  - Some instructions are more likely to result in a jump than others
  - Can get up to 75% success
- Taken/Not taken switch
  - Based on previous history
  - Good for loops
  - Refined by two-level or correlation-based branch history
- Correlation-based
  - In loop-closing branches, history is good predictor
  - In more complex structures, branch direction correlates with that of related branches
  - Use recent branch history as well
- Delayed Branch
  - Do not take jump until you have to
  - Rearrange instructions

Dealing With Branches

It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired.

That's All

Thank You