Computer Architecture

Lecture 11

Processor Structure & Function

Md. Biplob Hosen
Lecturer, IIT-JU

Reference Books

- Computer Organization and Architecture: Designing for Performance- William Stallings (8^{th} Edition)
  - Any later edition is fine

Instruction Pipelining

- Similar to the use of an assembly line in a manufacturing plant
- An instruction has a number of stages

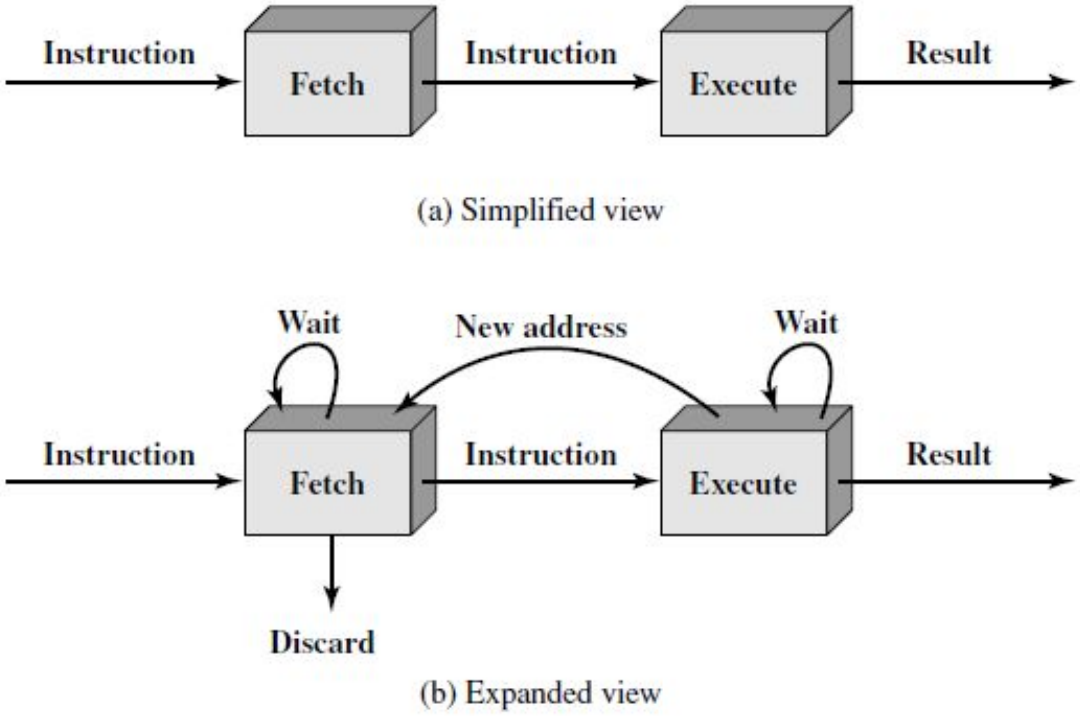


Figure: Two-Stage Instruction Pipeline

Instruction Pipelining

- This pipeline has two independent stages
  - The first stage fetches an instruction and buffers it
  - When the second stage is free, the first stage passes it the buffered instruction
- While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction
- This is called *instruction prefetch* or *fetch overlap*

Instruction Pipelining

- Speed-up is unlikely for two reasons:
- The execution time will generally be longer than the fetch time
  - Fetch stage may have to wait for some time before it can empty its buffer
- A conditional branch instruction makes the address of the next instruction to be fetched unknown
  - Fetch stage must wait until it receives the next instruction address from the execute stage
- The execute stage may then have to wait while the next instruction is fetched

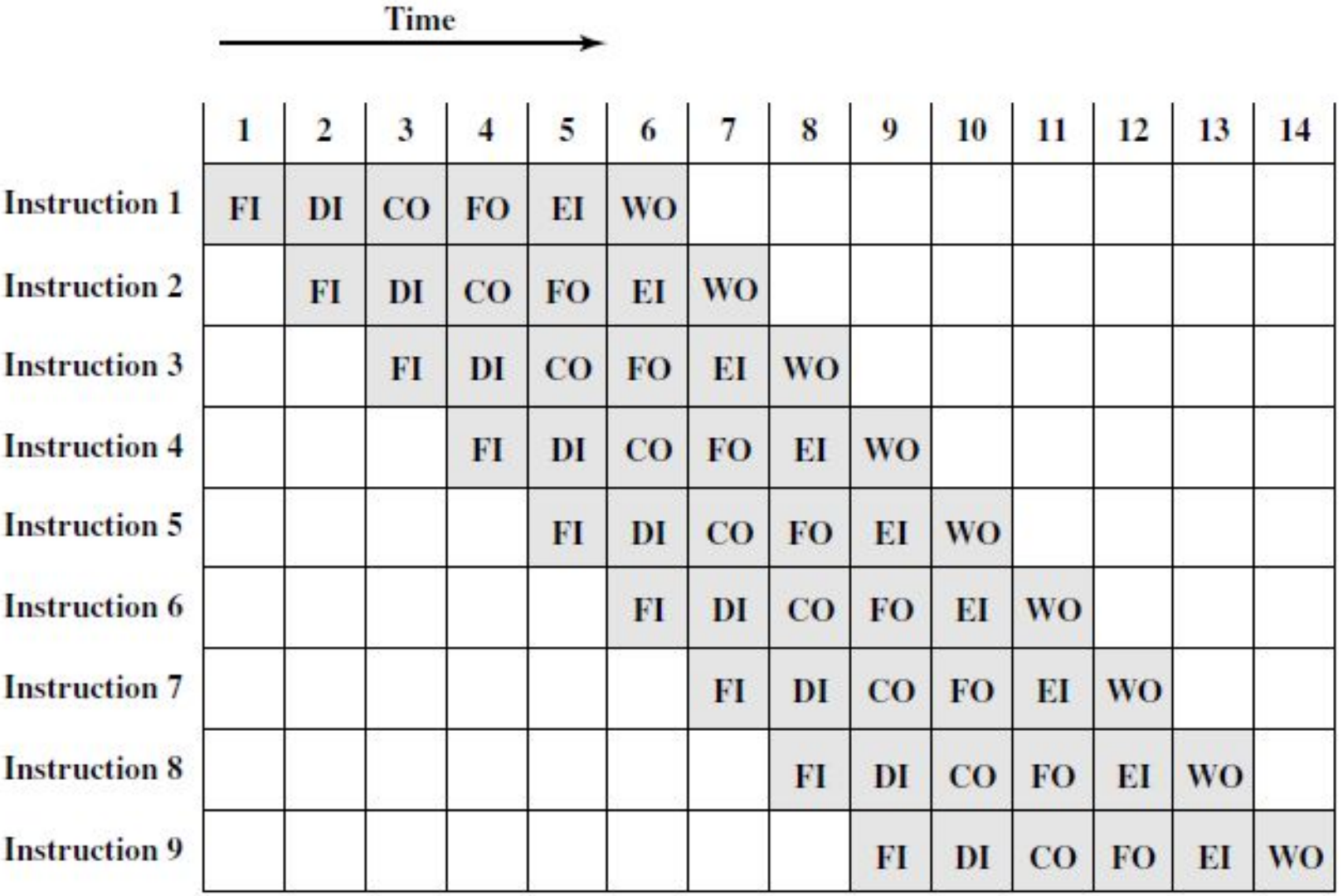
Instruction Pipelining

- Solution for problem 2:
- When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction
  - Then, if the branch is not taken, no time is lost
  - If the branch is taken, the fetched instruction must be discarded and a new instruction fetched

Instruction Pipelining

- Stages of instruction processing
  - Fetch instruction (FI): Read the next expected instruction into a buffer
  - Decode instruction (DI): Determine the opcode and the operand specifiers
  - Calculate operands (CO): Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation
  - Fetch operands (FO): Fetch each operand from memory. Operands in registers need not be fetched
  - Execute instruction (EI): Perform the indicated operation and store the result,if any, in the specified destination operand location
  - Write operand (WO): Store the result in memory

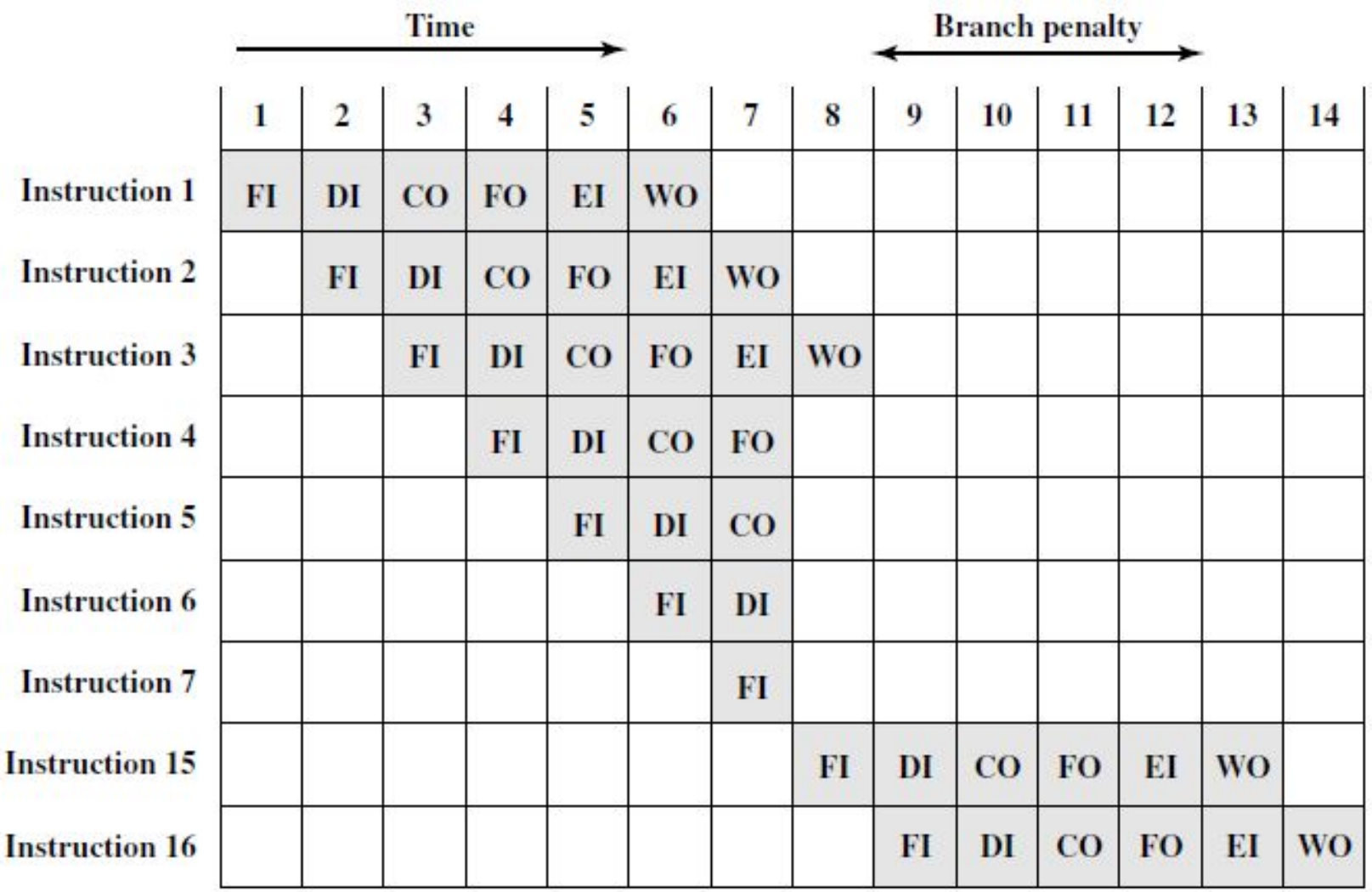
Instruction Pipelining



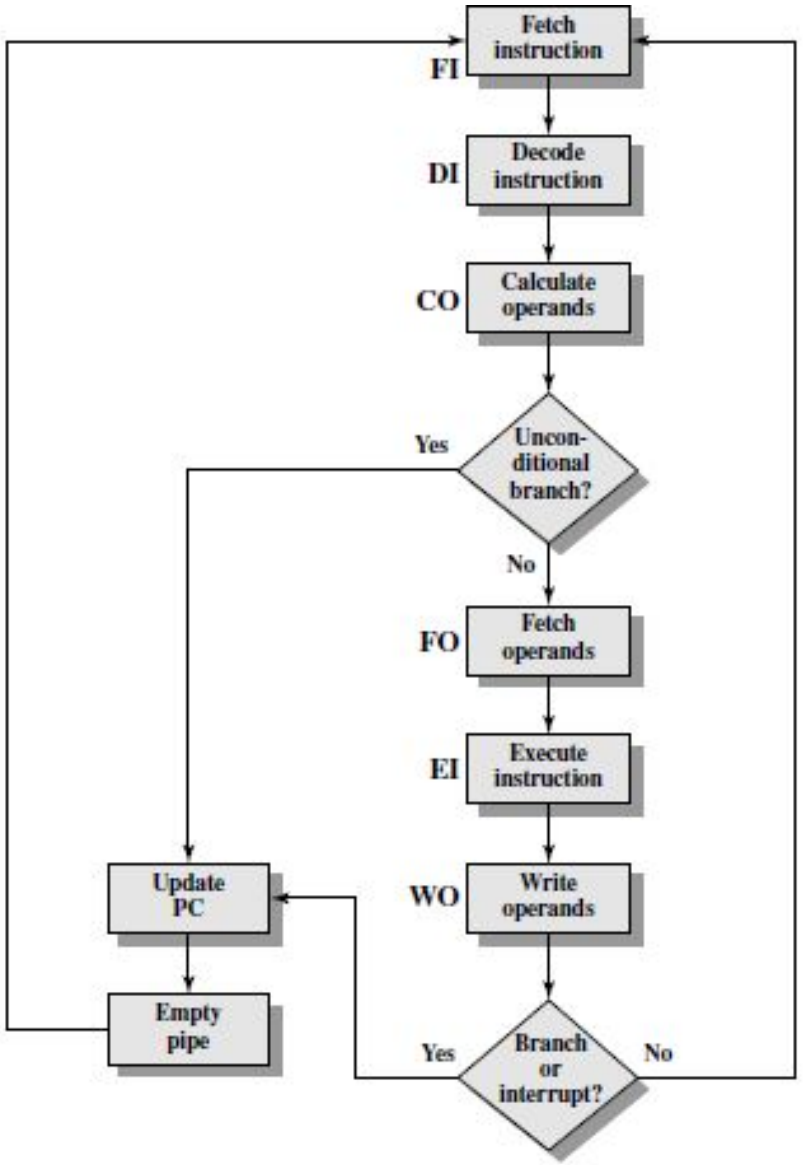
Instruction Pipelining

- Several other factors limit the performance
  - If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages
  - Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches

Instruction Pipelining



Instruction Pipelining



Flowchart:

Six-Stage CPU Instruction Pipeline

Instruction Pipelining

- Though it might appear that the greater the number of stages in the pipeline, the faster the execution rate
- At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions
  - This overhead can appreciably lengthen the total execution time of a single instruction
- This is significant when sequential instructions are logically dependent, either through heavy use of branching or through memory access dependencies
- The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages
- Another consideration is latching delay: It takes time for pipeline buffers to operate and this adds to instruction cycle time

Pipeline Performance

- The cycle time of an instruction pipeline is the time needed to advance a set of instructions one stage through the pipeline
- The cycle time can be determined as:

$$\tau = \max_i [\tau_i] + d = \tau_m + d \qquad 1 \leq i \leq k$$

- Where:
- τ_i = time delay of the circuitry in the ith stage of the pipeline
- τ_m = maximum stage delay (delay through stage which experiences the largest delay)
- k = number of stages in the instruction pipeline
- d = time delay of a latch, needed to advance signals and data

Pipeline Performance

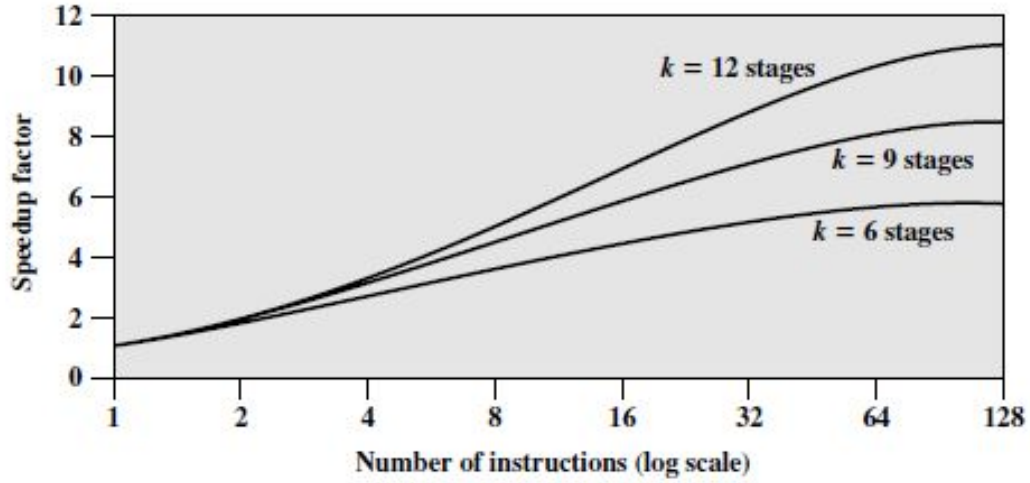
- Time delay, *d* is equivalent to a clock pulse and τ_m >>d
- Now suppose that n instructions are processed, with no branches
- Let $T_{k,n}$ be the total time required for a pipeline with k stages to execute n instructions

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

- A total of k cycles are required to complete the execution of the first instruction, and the remaining n-1 instructions require n-1 cycles
- Consider a processor with equivalent functions but no pipeline, and assume that the instruction cycle time is $k\tau$

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

Pipeline Performance



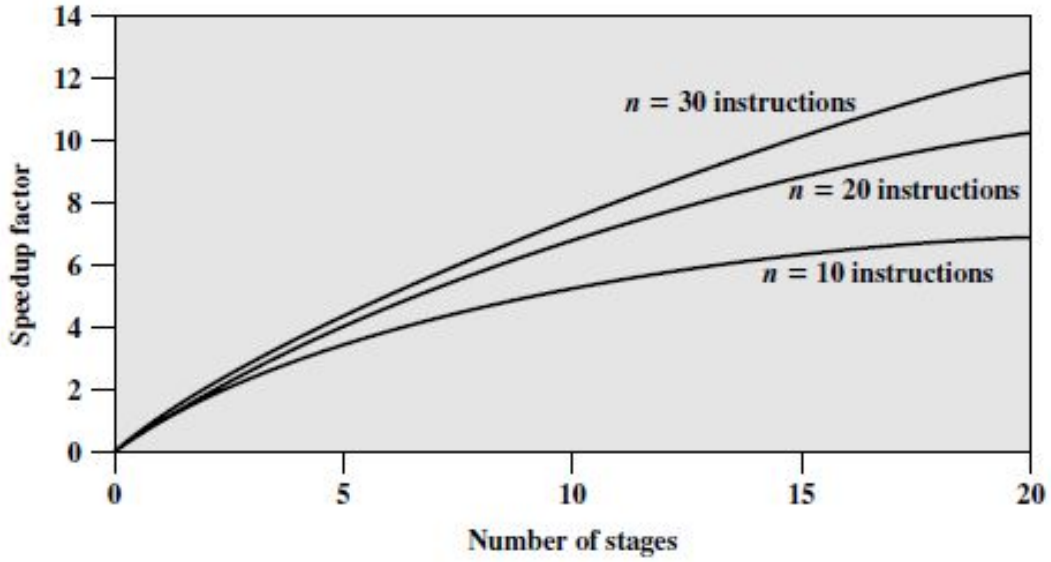


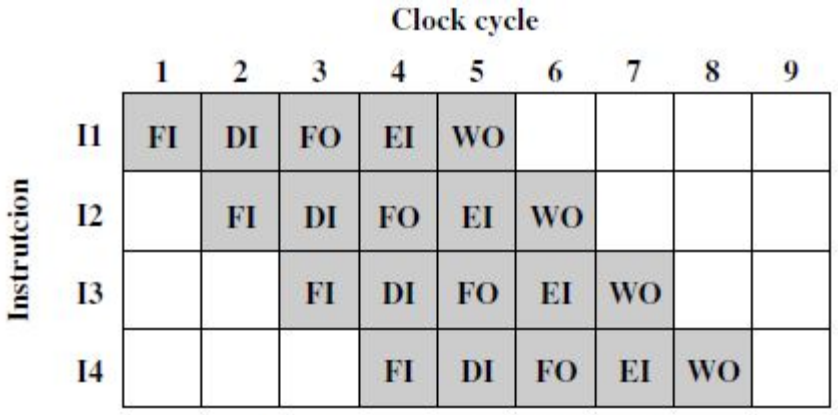
Figure: Speedup Factors with Instruction Pipelining

Pipeline Hazards

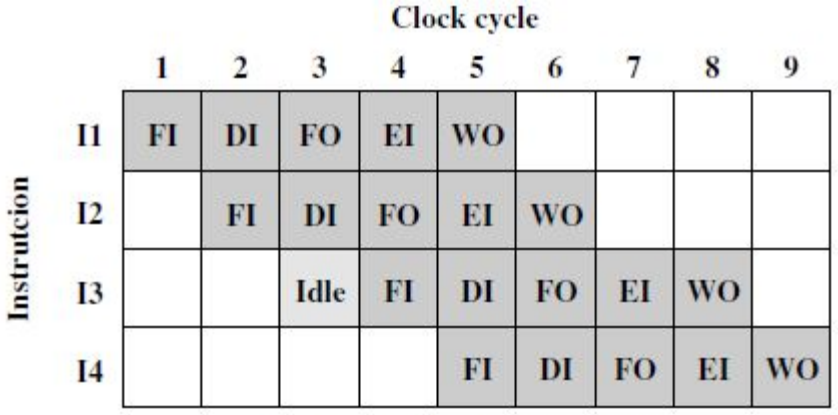
- A pipeline hazard occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution
- Referred to as also a *pipeline bubble*
- There are three types of hazards:
  - Resource
  - Data and
  - Control

Pipeline Hazards

- Resource Hazards: A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource
- The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline
- Sometime referred to as a structural hazard



(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

Pipeline Hazards

- What happens when multiple instructions are ready to enter the execute instruction phase and there is a single ALU?

- Read the book!

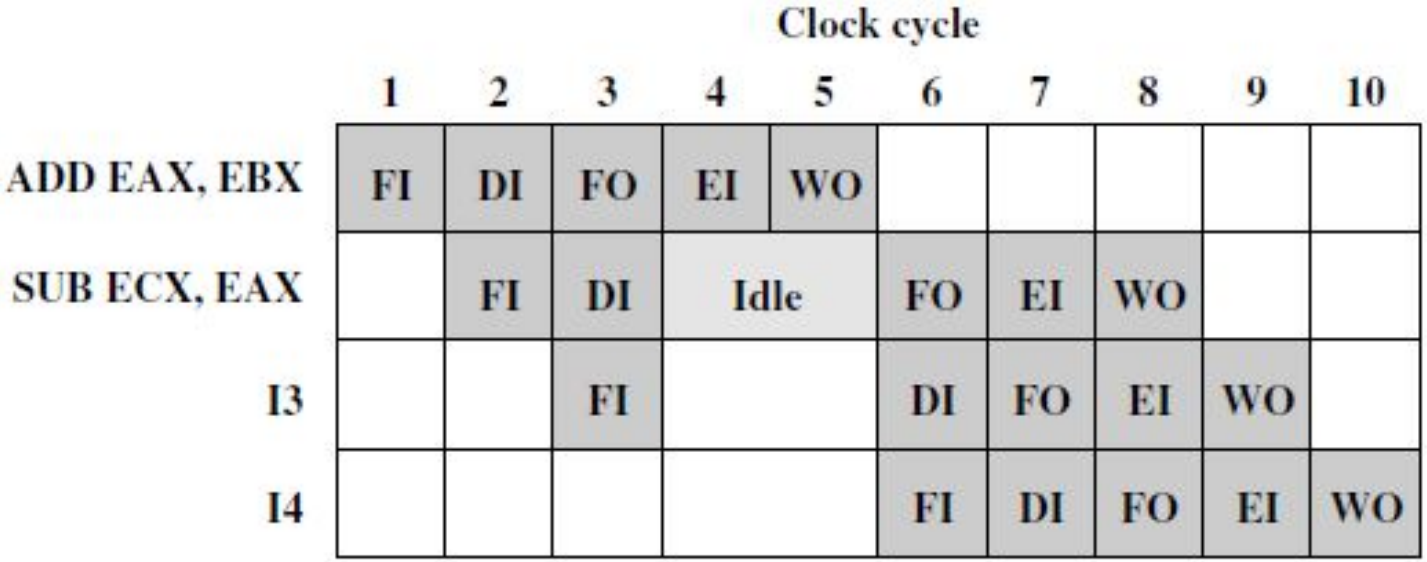
Pipeline Hazards

- Data Hazards: Occurs when there is a conflict in the access of an operand location
- Two instructions in a program are to be executed in sequence and both access a particular memory or register operand
- If the two instructions are executed in strict sequence, no problem occurs
- However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution
- In other words, the program produces an incorrect result because of the use of pipelining

Pipeline Hazards

- Three types of data hazards
- Read after write (RAW), or true dependency: An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location
  - A hazard occurs if the read takes place before the write operation is complete
- Write after read (WAR), or anti-dependency: An instruction reads a register or memory location and a succeeding instruction writes to the location
  - A hazard occurs if the write operation completes before the read operation takes place
- Write after write (WAW), or output dependency: Two instructions both write to the same location
  - A hazard occurs if the write operations take place in the reverse order of the intended sequence

RAW Data Hazard



Pipeline Hazards

- Control Hazards: Also known as a branch hazard, occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded

Dealing with Branches

- Multiple Techniques:
  - Prefetch branch target
  - Loop buffer
  - Branch prediction
  - Delayed branch

Dealing with Branches

- Multiple Streams: A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice
- A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams
- There are two problems with this approach:
  - With multiple pipelines there are contention delays for access to the registers and to memory
  - Additional branch instructions may enter the pipeline (either stream) before the original branch decision is resolved

Dealing with Branches

- Prefetch Branch Target: When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch
- This target is then saved until the branch instruction is executed
- If the branch is taken, the target has already been prefetched
- LOOP BUFFER : A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the n most recently fetched instructions, in sequence

Dealing with Branches

- If a branch is to be taken, the hardware first checks whether the branch target is within the buffer
- If so, the next instruction is fetched from the buffer
- The loop buffer has three benefits:

1) With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address
   - Thus, instructions fetched in sequence will be available without the usual memory access time

Dealing with Branches

2) If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer
   - This is useful for the rather common occurrence of IF–THEN and IF–THEN–ELSE sequences
3) This strategy is particularly well suited to dealing with loops, or iterations; hence the name *loop buffer*
   - If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration
   - For subsequent iterations, all the needed instructions are already in the buffer

Dealing with Branches

- Branch Prediction: Multiple techniques
  - Predict never taken
  - Predict always taken
  - Predict by opcode
  - Taken/not taken switch
  - Branch history table
- Delayed Branch: It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired

Thank you!