Instruction-level parallelism (ILP):

- Pipelining became universal technique in 1985.
- Instruction-level parallelism (ILP) is a measure of how many of the instructions in a computer program can be executed simultaneously.

ILP must not be confused with concurrency:

- ILP is the parallel execution of a sequence of instructions belonging to a specific thread of execution of a process (A running program with its set of resources: address space, a set of registers, its identifiers, its state, program counter (aka instruction pointer), and more).
- Concurrency involves the assignment of threads of one or different processes to a CPU's core in a strict alternation, or in true parallelism if there are enough CPU cores, ideally one core for each runnable thread.

Two main approaches:

- Dynamic
    - Hardware-based
    - Used in server and desktop processors
    - The processor decides at run time which instructions to execute in parallel
- Static
    - Compiler-based
    - The compiler decides which instructions to execute in parallel
    - Not as successful outside of scientific applications

Consider the following program:

1. e = a + b
2. f = c + d
3. m = e * f

- Operation 3 depends on the results of operations 1 and 2. It cannot be calculated until both of them are completed.
- Operations 1 and 2 do not depend on any other operation, so they can be calculated simultaneously.

Assumption: Each operation can be completed in one unit of time.

- If operation 1 and 2 can be done simultaneously, total time required is 2 units, otherwise, 3 units.

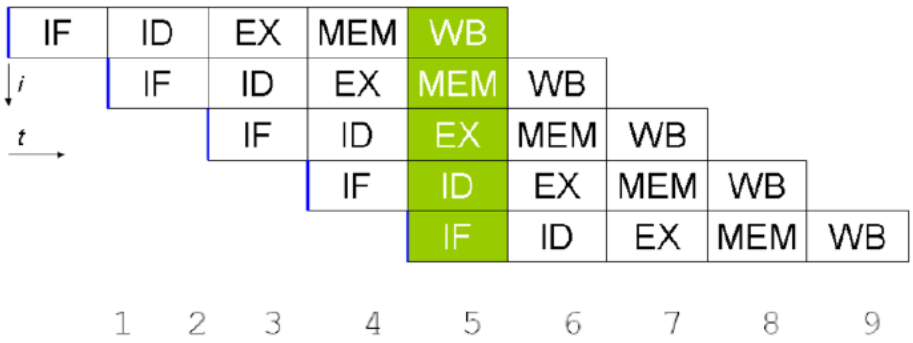
Pipelining: Each instruction is split up into a sequence of steps – different steps can be executed concurrently by different circuitry.

RISC	CISC
• No Direct Memory Access • Load and Store instructions are required for memory access. • Same size of the instructions.	• Direct Memory Access • Load and Store instructions are NOT required for memory access. • Different sizes of the instructions.

a = b + c
LD r1, b
LD r2, c
ADD r1, r1, r2
ST a, r1

Basic pipeline in a RISC processor:
- IF – Instruction Fetch
- ID – Instruction Decode
- EX – Instruction Execution
- MEM – Memory Access
- WB – Register Write Back

Techniques:
- Instruction pipelining where the execution of multiple instructions can be partially overlapped.
- Superscalar execution, VLIW, and the closely related explicitly parallel instruction computing concepts, in which multiple execution units are used to execute multiple instructions in parallel.



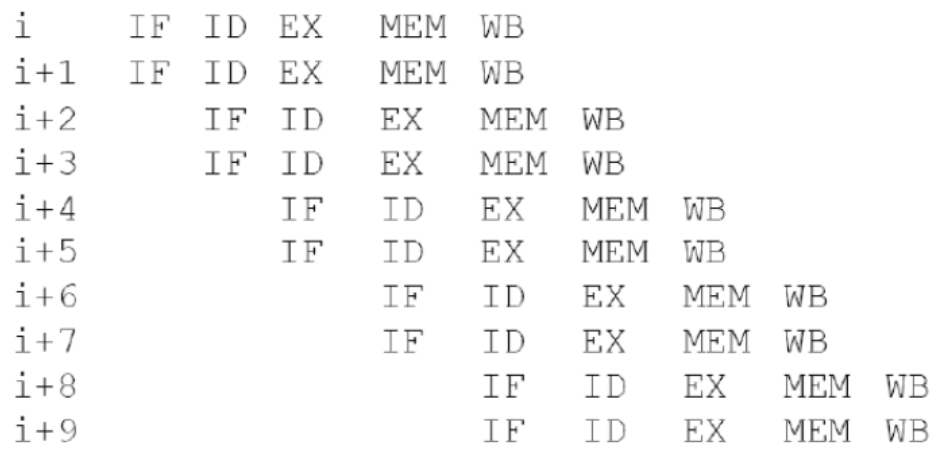
With parallel, for a program with 10 instructions, total time is 14 units; Otherwise 50 units.

```
12345
 12345
  12345
   12345
    12345
     12345
      12345
       12345
        12345
         12345
```

Assumption: We can have

- 2 IF in parallel
- 2 ID in parallel
- 2 EX in parallel
- 2 MEM in parallel
- 2 WB in parallel

Time for 10 instructions will be 9



CPI = Cycles Per Instruction
Pipeline CPI = Ideal pipeline CPI + Structural stalls + RAW (Read After Write) stalls + WAR (Write After Read) stalls + WAW (Write After Write) stalls + Control stalls

Classes of pipeline hazards:

- 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 (Program Counter).

$CPI_{ideal} = 1$
CPI_{stalls} = Branches, Exceptions, Memory Latency, Cache Misses, Data Dependence etc

Overheads include:

- Setup and Hold Times
- Inequality in work per stage

Pipelining:

- Running Time = Instructions * Avg. CPU Cycles per Instruction * Time per Cycle

Goal: Decrease running time - Reduce one or more of these THREE

- Instructions depends upon Processor, Compiler and Assembly Language of the processor
- Pipelining lowers time per cycle by dividing sequence of instructions into sequence of steps (each requiring less time)

Q: How fast can a program be run on a processor with instruction-level parallelism?
A: Depends on:

- The potential parallelism in the program.
- The available parallelism on the processor.
- Our ability to extract parallelism from the original sequential program.
- Our ability to find the best parallel schedule given scheduling constraints.

Traditional register-allocation techniques aim to minimize the number of registers used when performing a computation.

EXAMPLE:

```
LD t1, a   // t1 = a Load from Memory into Register
ST b, t1   // b = t1 Store into Memory from register b = a
LD t2, c   // t2 = c Load from Memory into Register
ST d, t2   // d = t2 Store into Memory from register d = c
```

t1 and t2 are two pseudoregisters.

Q. Can one be used?
A. YES !

```
LD t1, a   // t1 = a Load from Memory into Register
ST b, t1   // b = t1 Store into Memory from register b = a
LD t1, c   // t1 = c Load from Memory into Register
ST d, t1   // d = t1 Store into Memory from register d = c
```