Pipelining

Basic concept: Assembly line

- Split a job A into n sequential subjobs $(A_1, A_2, ..., A_n)$ with each A_i taking approximately the same time
- Each subjob is processed by a different substation (resource) or, equivalently passes through a series of *stages*
- Overlap of subjobs of different jobs, i.e.: when subjob A_1 of job A is finished on stage 1, subjob A_2 of job A starts on stage 2 while subjob B_1 of job B starts on stage 1.

Pipelining – Performance

Execution time of a single job:

- can be longer since each stage takes the same time and that time is the longest time for any of the substages

But throughput is enhanced since:

- A new job can start at every stage time, and it follows that one job is completed at every stage time

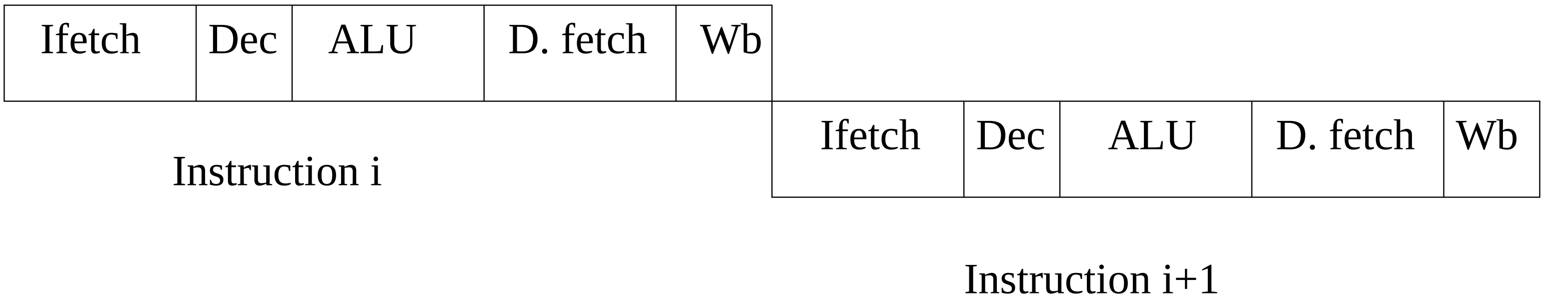
In the best case, throughput is n times more if there are n stages. But, this is a little too optimistic because

- Execution time of a job by itself could be less than n stage times
- We have assumed that the pipeline could be kept full all of the time

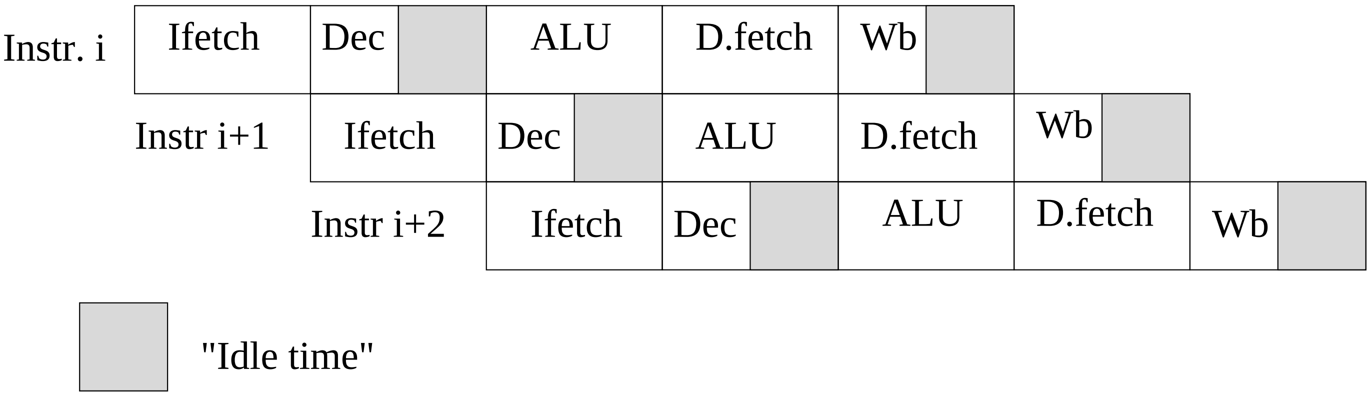
Nonetheless, shows great promise!

Pipelining applied to instruction execution

Recall the 5 steps of instruction execution (multiple cycle implementation)



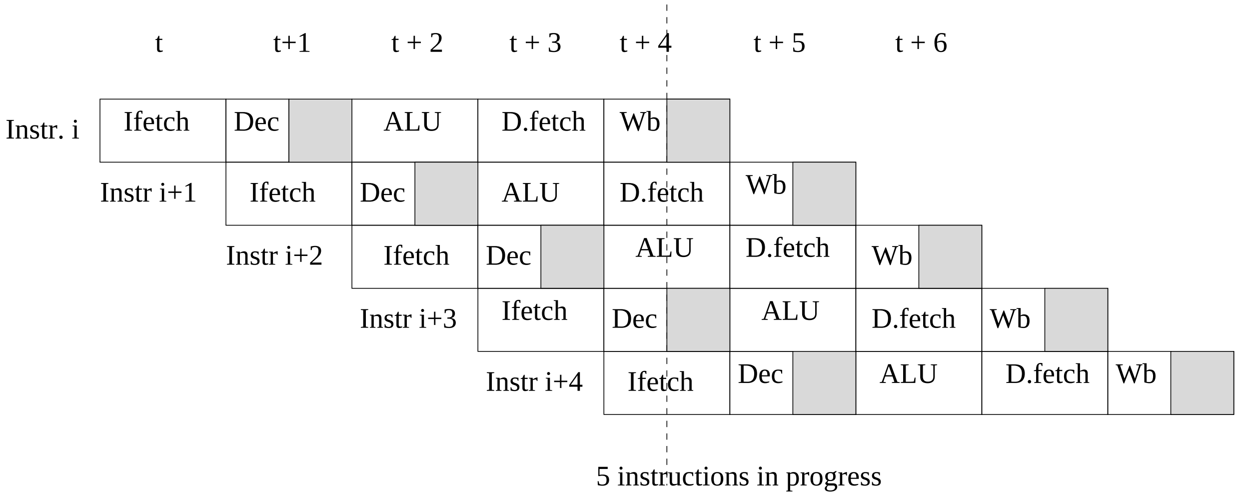
In pipelined mode



Pipeline parallelism

Note that a given time, 5 instructions are in progress.

Note also that an individual instruction takes longer



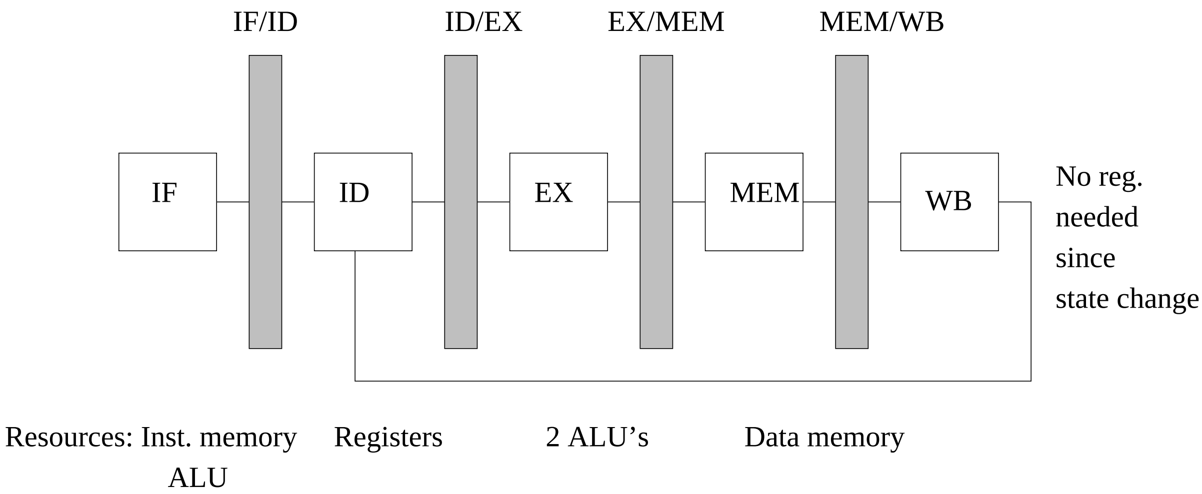
Pipeline implementation requirements

At a given time, all 5 stages are active (on different instructions)
Thus we need all the resources that were used in a single cycle implementation

All stages are independent and isolated from each other. All information gathered or kept at stage i that will be needed at stage $i+1$ and after must be passed on in "pipeline registers" that separate each stage. For example:

- The register number where the result of the ALU will be stored
- The register number where data will come from in a store instruction
- The 16 bit offset or immediate value
- The updated PC
- The contents of the register that will be written to memory in a store instruction
- etc.

Pipeline datapath (highly abstracted)



What could go wrong? Hazards.

Structural hazards. Two instructions want to use the same resource (at different stages). This would happen if we had a single memory for data and instructions. (In real systems, this is taken care of by separate instruction and data caches – see later).

Control hazards. This happens when a branch is taken. Since the decision cannot occur before the next instruction in sequence has already been fetched, we need to take some remedial action to be described soon. We will talk briefly also about *speculative execution.*

Data hazards. This happens when instructions that are in the pipeline at the same time are *dependent*, i.e., the result of an instruction is needed for the next instruction. We'll see that in some cases we can still proceed using *bypassing* while in others the pipeline will have to be *stalled*.

Pipeline datapath (a little less abstracted)

Let's look back at Figure 5.17 (single cycle execution) with delimiters between the resources used at each instruction step