Pipelining

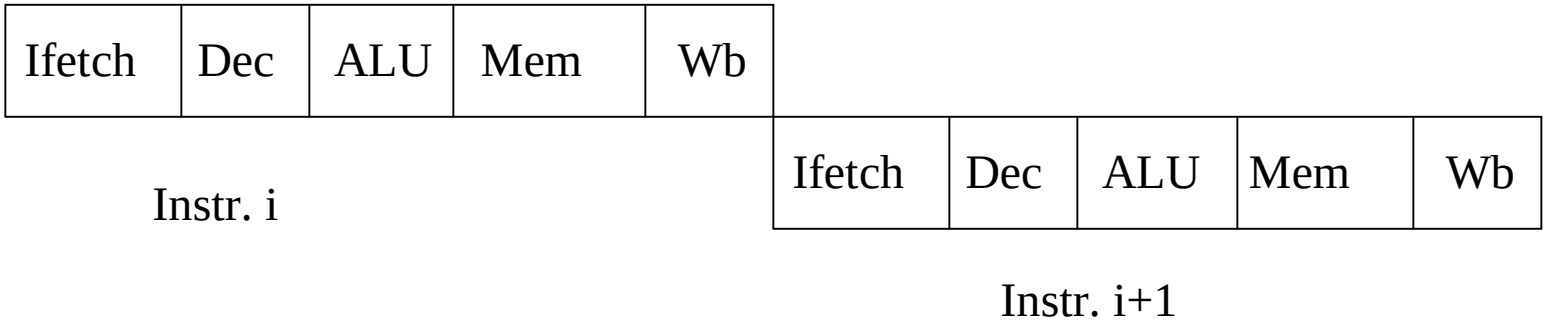
- Basic concept of assembly line
  - Split a job *A* into *n* sequential subjobs (A_1,A_2,…,A_n) with each A_i taking approximately the same time
  - Each subjob is processed by a different substation (or resource) or, equivalently, passes through a series of *stages*
  - Subjobs of different jobs overlap their execution, i.e. when subjob A_1 of job *A* is finished in stage 1, subjob A_2 of job *A* will start executing in stage 2 while subjob $\boldsymbol{B}_1$ of job *B* will start executing in stage 1

Pipeline performance

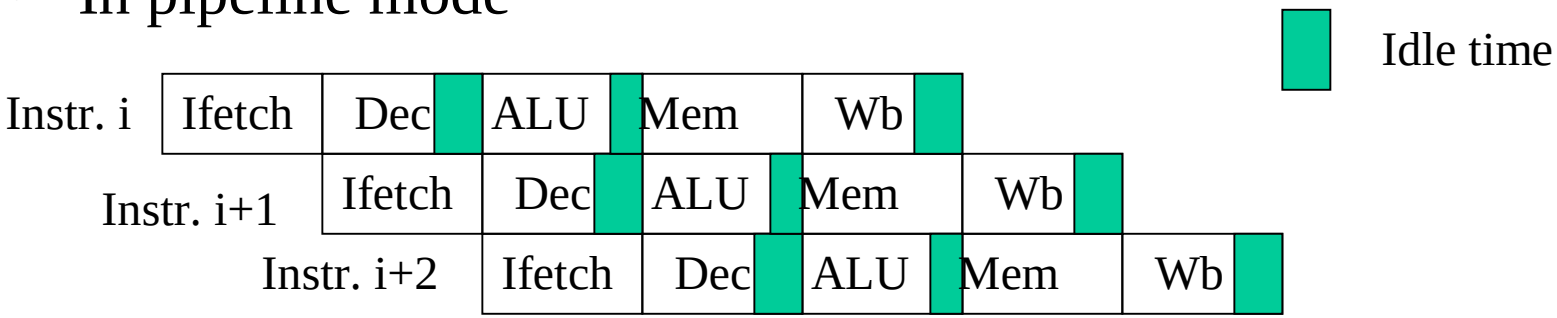
- Latency of a single job can be longer
  - because each stage takes as long as the longest one, say t_{max}
  - because all jobs have to go through all stages even if they don't do anything in one of the stages
- Throughput is enhanced
  - Ideally, in steady state, one job completes every t_{max} rather than after (t_1 + t_2 + ... t_n)
- In the ideal case (all t_i are the same) throughput is *n* times better if there are *n* stages. But :
  - Execution time of a job could take less than n stages
  - We assume that the pipeline can be kept full all the time

Pipeline applied to instruction execution

- Multiple cycle implementation

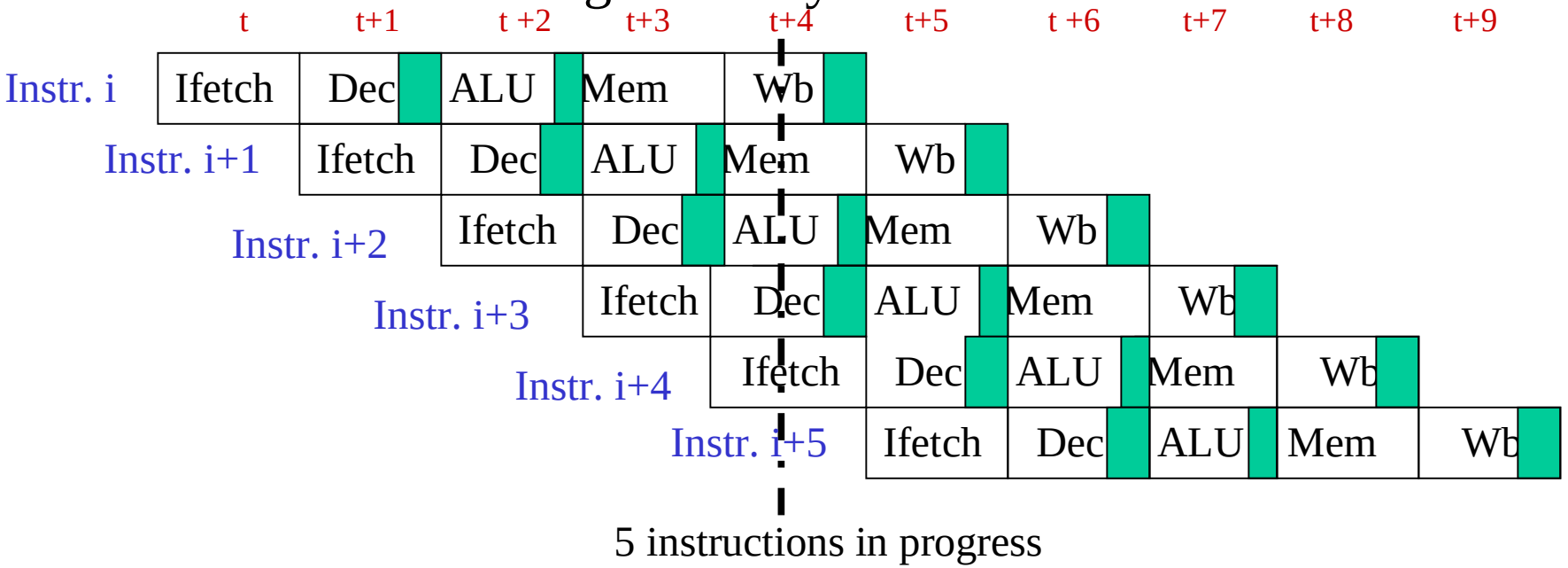


- In pipeline mode



Pipeline parallelism

- Note that at any given time after the pipeline is full, we have 5 instructions in progress; however each individual instruction has a longer latency



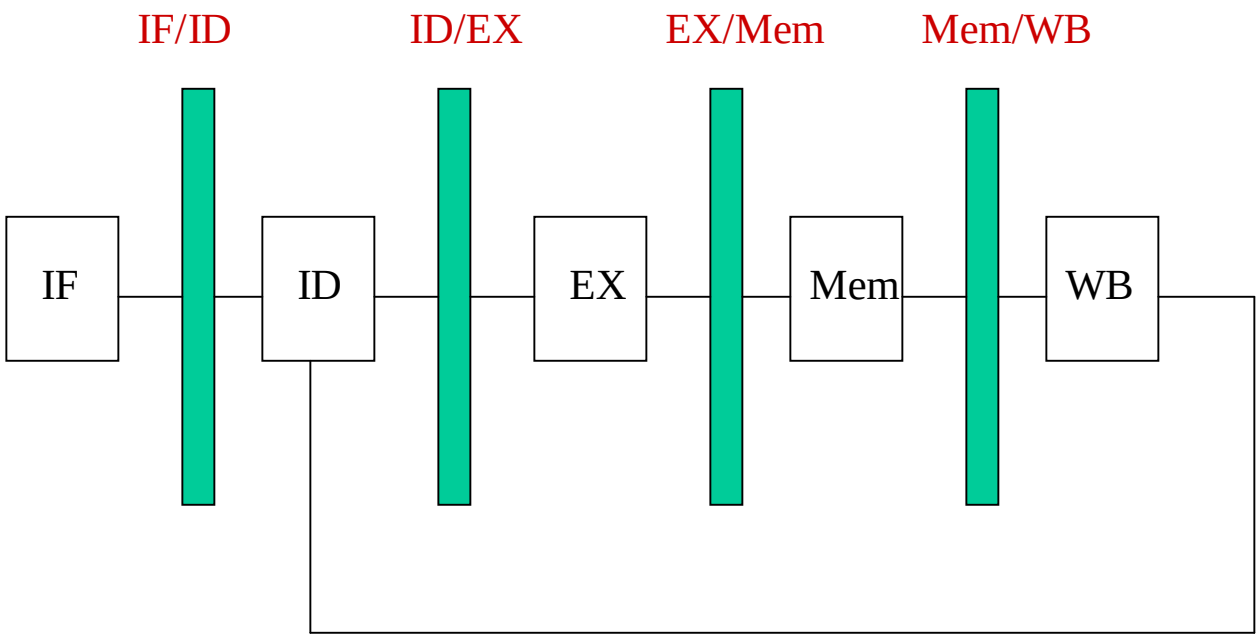
Pipeline implementation requirements

- 5 stages are active on 5 different instructions
- Thus all the resources needed for single cycle implementation will be required (at least)
- All stages are independent and isolated from each other. This implies that all information gathered at stage *i* needed in stages *i+1*, *i+2 etc.*…must be kept and passed from stage to stage.
- For that purpose we will use *"pipeline registers"* (or latches) between each stage

Example of what to store in pipeline registers

- The following is **not** an exhaustive list (just a sample)
- The register number where the result will be stored
  - Known at stage 2; needed at stage 5
- The register number of the data containing the contents of a “store” as well as the contents of that register
  - Known at stage 2; needed at stage 4
- The immediate value
  - Known at stage 2; needed at stage 3
- The updated PC
- etc ..

Pipeline data path (highly abstracted)



Pipeline data path (a little less abstracted)

- Let's look back at Fig. 5.17 (Single cycle implementation) and see where the pipeline registers should be put and what additional resources, if any, are needed.

Hazards

- Hazards are what prevents the pipeline to be ideal
- Three types of hazards:
  - *Structural* where two instructions at different stages want to use the same resource. Solved by using more resources (e.g., instruction and data memory; several ALU's)
  - *Data hazards* when an instruction in the pipeline is *dependent* on another instruction still in the pipeline. Some of these hazards will be resolved by *bypassing* and some will result in the pipeline being *stalled*
  - *Control hazards* which happens on a successful branch (or procedure call/return). We'll investigate several techniques to take care of these hazards.

Notation (cf. Figure 6.18)

- Pipeline registers keeping information between stages labeled by the names of the stages they connect (IF/ID, ID/EX, etc.)
- Information flows from left to right except for
  - writing the result register (potential for data hazard)
  - modifying the PC (potential for control hazard)

Tracing one instruction through all 5 stages (St 1)

- *Stage 1*: Instruction fetch and increment PC (same for all instructions)
  - Instruction fetched from instruction memory. The instruction will be needed in subsequent stages. Hence stored in IF/ID
    - recall (i) the IR register in multiple cycle implementation, and (ii) that in single cycle implementation we had two "memories", one for instruction and one for data
  - PC ← PC + 4
    - The incremented PC will be needed to fetch the next instruction at the next cycle but it will also be needed if we have to do branch target computation. Hence incremented PC saved in IF/ID
  - Resources needed: Instruction memory and ALU
  - IF/ID contains instruction and incremented PC

Tracing one instruction through all 5 stages (St 2)

- *Stage 2:* Instruction decode and register read (same for all instructions)
  - We save in ID/EX everything that can be needed in next stages:
    - The instruction and the incremented PC (from IF/ID) (e.g., function bits can be needed, the name of the register to be written etc.)
    - The contents of registers rs and rt (recall registers A and B)
    - The sign extended immediate field (for imm. Inst., load/store, branch)
    - Note that we do not compute the branch target address here (why?).
  - Resources needed: register file
  - ID/EX contains PC, instruction, contents of rs and rt, extended imm. field

Tracing one instruction through all 5 stages (St 3)

- *Stage 3:* depends on the type of instruction being executed. Let's assume a Load. Hence this stage is "address computation"
  - ALU sources come from ID/EX (rs and immediate field)
  - ALU result stored in EX/Mem
  - Contents of rs, rt, and imm. field not needed any longer
  - Looks like PC not needed either but we'll keep it because of possible *exceptions* (see later in the course)
  - Need to keep in EX/Mem part of the instruction (name of the rt register) and the indication we have a load

Stage 3 (ct'd)

- Resource needed: ALU
- But if we had a branch we need two ALU's, one for the branch target computation and one for comparing registers.
- EX/Mem needs to keep
  - result of ALU (for load/store and arith. instructions)
  - name of result register (for load and arith. instructions)
  - contents of rs (for store)
  - PC (in case of exceptions)

Tracing one instruction through all 5 stages (St 4)

- *Stage 4:* Access data memory. Still assuming we have a load
  - Access data memory with address kept in EX/Mem
  - Keep result of load in Mem/WB
  - Resource needed: Data memory
  - Mem/WB needs to keep
    - Result of load if load instruction
    - Result of arith. Instr. if arith instr (from EX/Mem)
    - Name of result register
    - No need to keep PC any longer (no exception occurs in last stage)

Tracing one instruction through all 5 stages (St 5)

- *Stage 5:* Write back register (for load and arith)
  - Contents of what to write and where to write it in Mem/WB
  - Nothing to be kept
  - Resource needed: register file
  - Ah! But registers were also needed in Stage 2!
    - We allow writing registers in first part of cycle and reading registers in second part of cycle

Summary of requirements of ideal pipeline data path

- Stages 1 and 2: info to be kept is similar for all instructions
- The width of the pipeline registers must be such that it fits the “maximum” amount of info that will be needed
- Instructions (except branches, see soon) must pass through all stages even if nothing is done in that stage
- The “state of the machine” is modified only in the last stage (except for branches)