COMPUTER ORGANIZATION (IS F242)

LECT 41: PIPELINING

Single Cycle Vs Multi Cycle

- Single Cycle
  - CPI = 1
  - Cycle time may be very long
- Multi Cycle
  - Shorter Cycle time
  - Higher clock rates
  - Each instruction requires many cycles to complete
  - Average CPI = 3 to 5
- Can we have the best out of these two??

Analysis

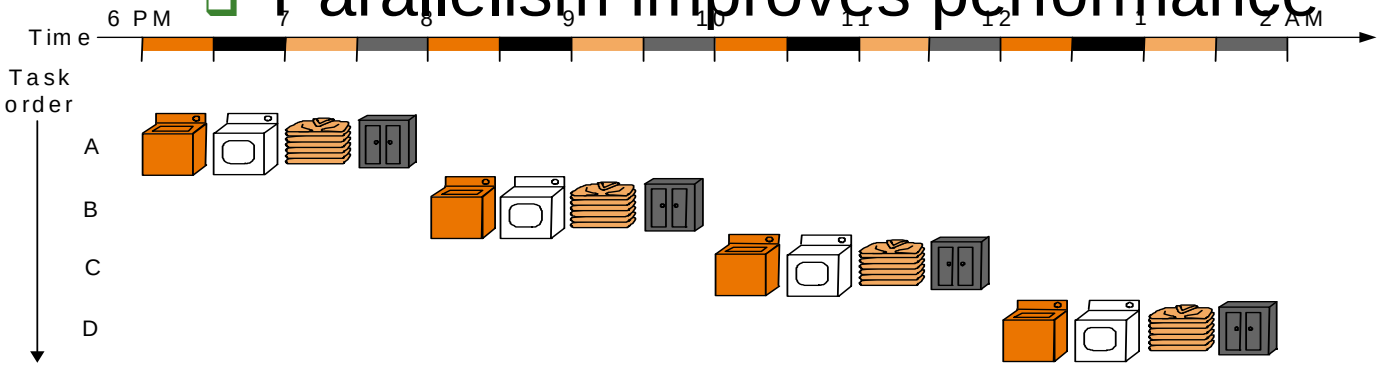
- Not all instructions need all five steps
- MCI uses different functional units assuming read and write are independent
- Many units are idle most of the execution time
- Lots of expensive hardware sitting IDLE without doing anything!!
- How to utilize them effectively??

Solution

- Overlap instructions
- Each step is a pipeline stage
- Performance
  - Pipeline throughput
  - Pipeline latency
    - Cycle time * Number of stages

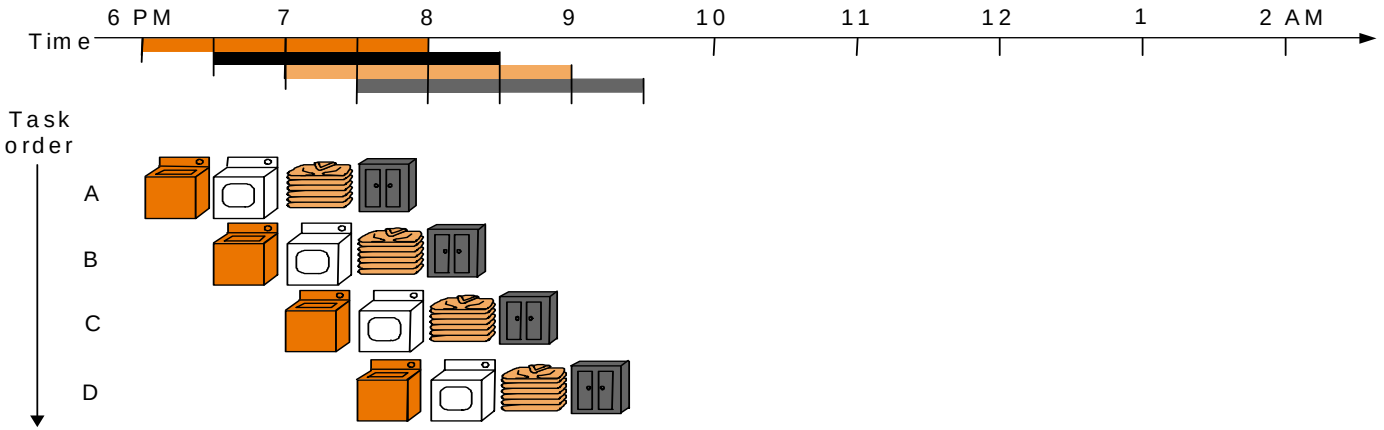
Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance



- Four loads:

  Speedup
  = 8/3.5 = 2.3



MIPS Pipeline

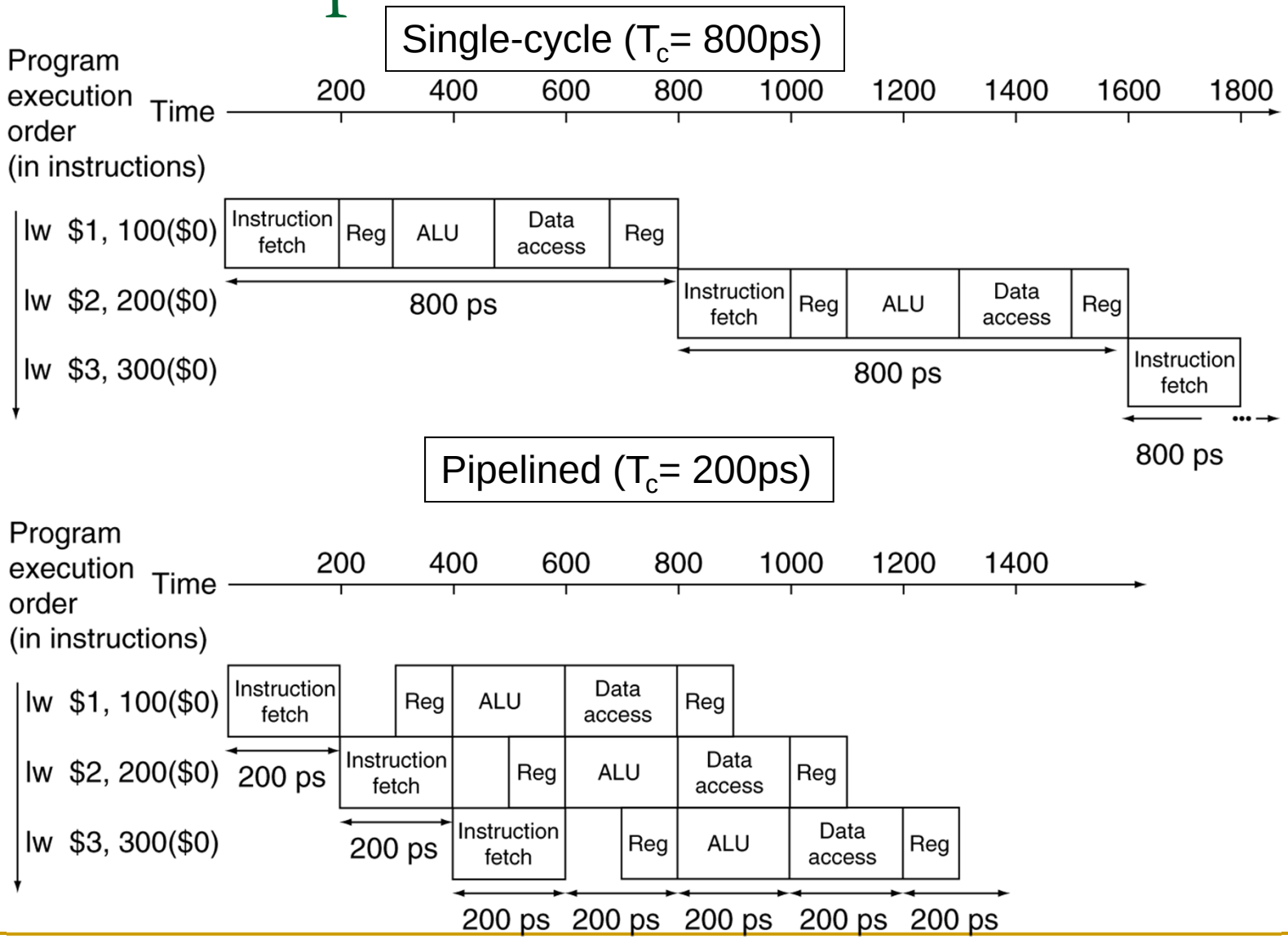
- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Pipeline Performance

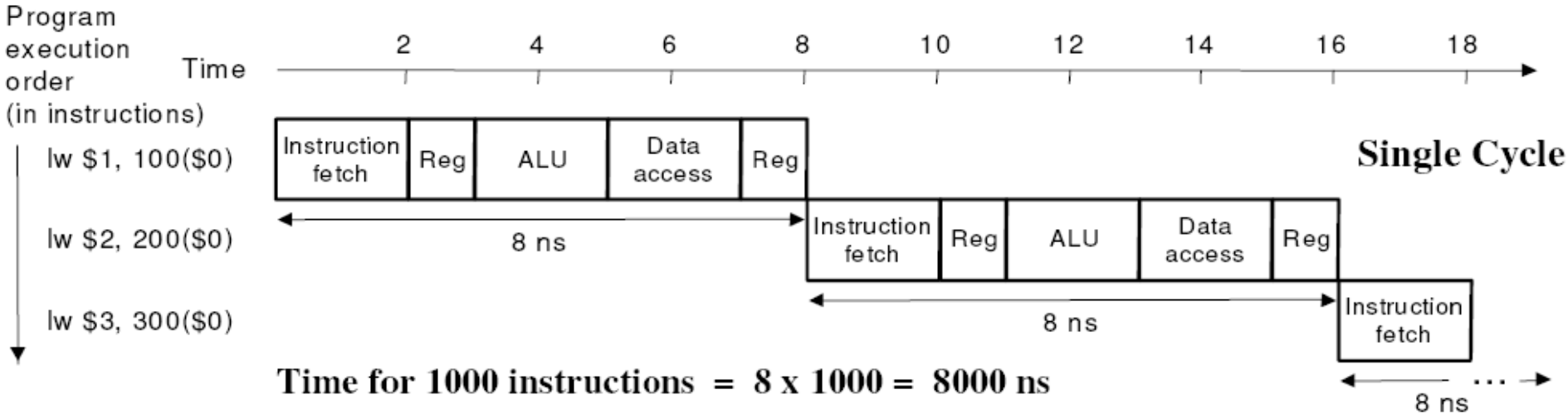
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

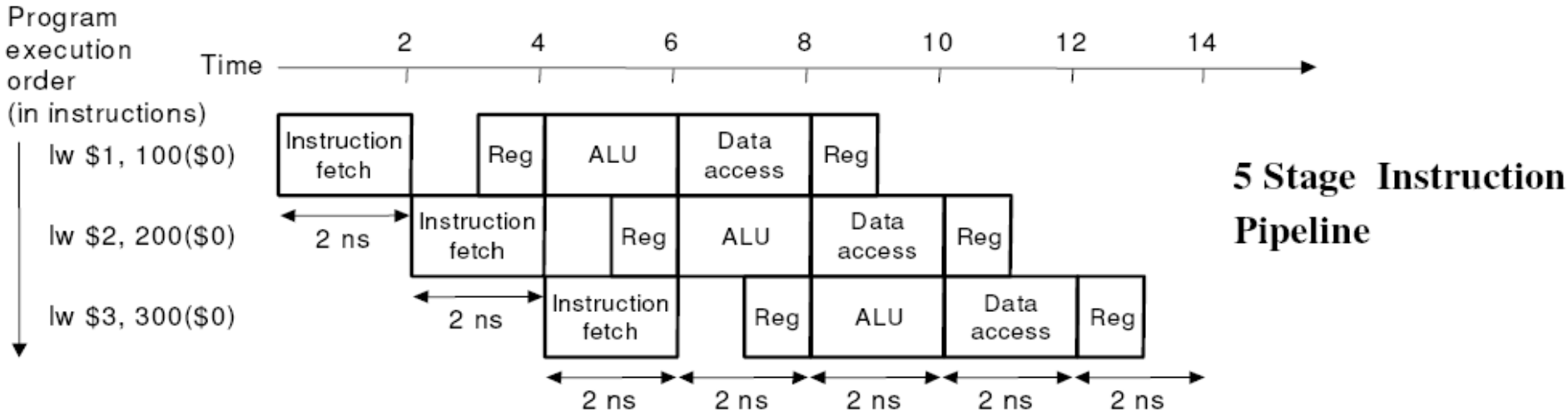
Pipeline Performance



Single Cycle Vs. Pipelining



Time for 1000 instructions = 8 x 1000 = 8000 ns



Time for 1000 instructions = time to fill pipeline + cycle time x 1000 = 8 + 2 x 1000 = 2008 ns

Pipelining Speedup = 8000/2008 = 3.98

Pipelining: Keep in Mind

- Pipelining *does not reduce latency* of a single task, it *increases throughput* of entire workload
- Pipeline rate *limited by longest stage*
  - *potential* speedup = number pipe stages
  - *unbalanced lengths* of pipe stages reduces speedup
- Time to *fill* pipeline and time to *drain* it – when there is *slack* in the pipeline – reduces speedup

Pipeline Performance

- Cycle time (t)
  - Time needed to advance a set of instructions one stage through the pipeline
  - t = Maximum stage delay + time delay of a latch
- Let n instructions are processed, with no branches.
- Total time required to execute n instructions for a pipeline with k stages

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

Pipeline Performance

- Speedup factor
  - **$S_k = nk/[k+(n-1)]$**
- How the value of K influence the speedup???
  - The larger the number of pipelines stages (k), the greater the potential for speedup.
  - However, as a practical matter, the potential gains of additional stages are countered by increases in cost, delays between stages.

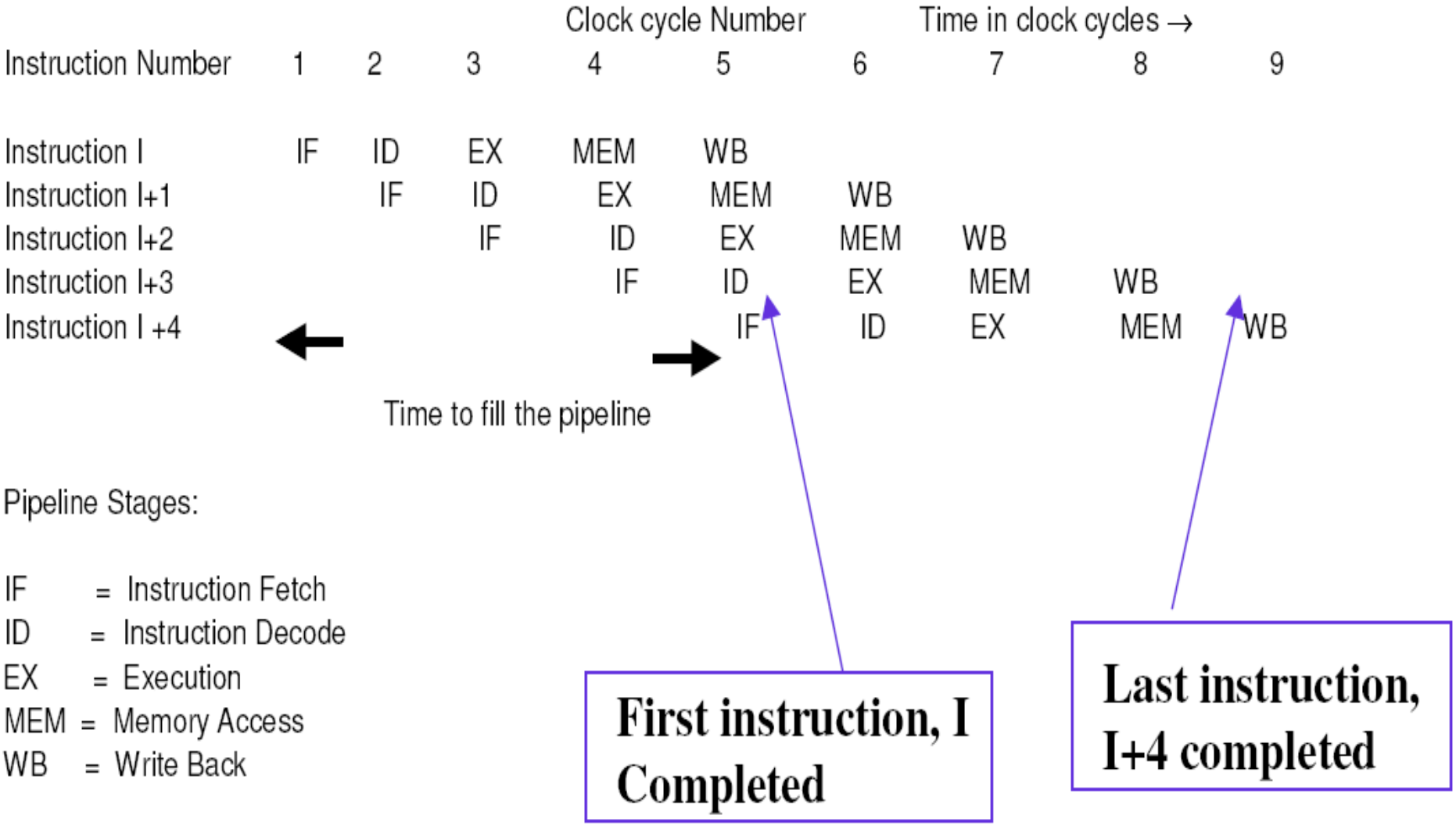
Pipeline Speedup

- The length of the machine cycle is determined by the time required by the slowest pipeline stage
- An important pipeline design consideration is to balance the length of each pipeline stage
- If all stages are perfectly balanced, then the time per instruction on a pipelined machine

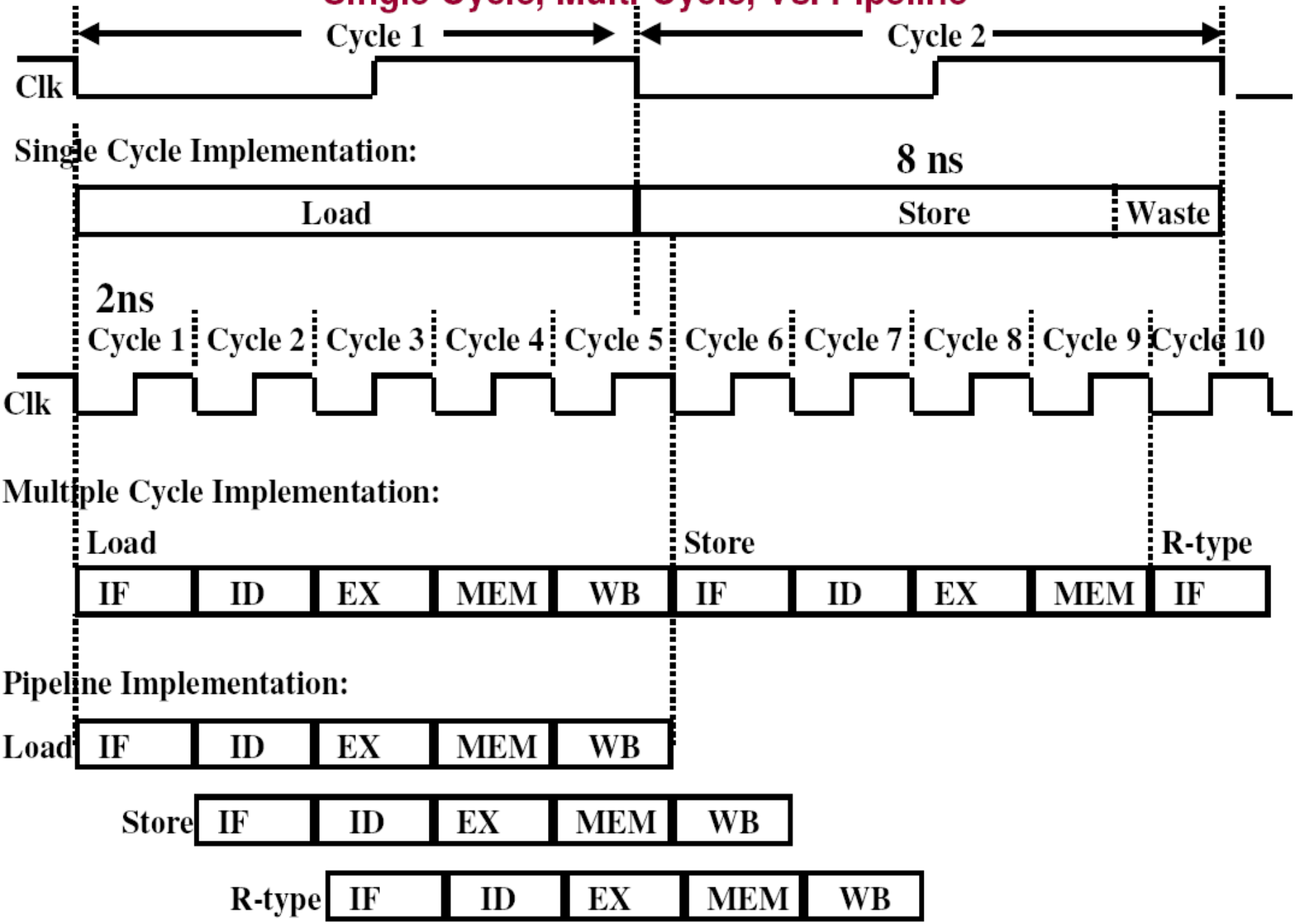
$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- Under Ideal conditions:
  - Speedup from pipeline = the number of pipeline stages = k
  - One instruction completes every cycle. CPI = 1

Pipelined Processing Representation



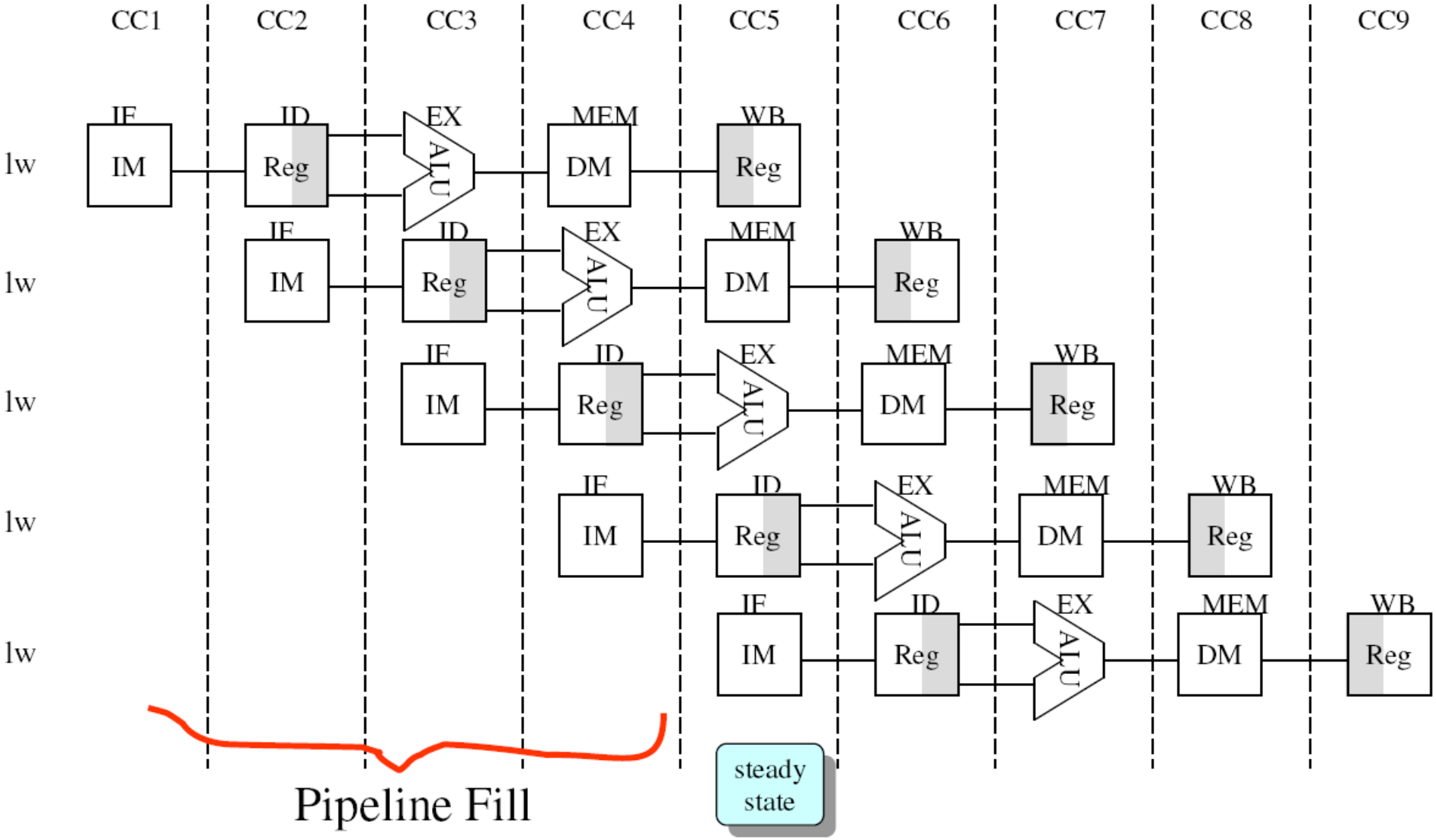
Single Cycle, Multi-Cycle, Vs. Pipeline



Performance Comparison

- For 1000 instructions, execution time:
  - Single cycle Machine:
    - (8ns/cycle) * (1 CPI) * (1000 inst) = 8000ns
  - Multi Cycle Machine:
    - (2ns/cycle) * (4.03 CPI) * (1000 inst) = 8060ns
  - Ideal Pipelined machine, 5 stages:
    - (2ns/cycle) * [(1 CPI*1000 inst) + 4 cycle fill] = 2008ns

LW - Execution in a Pipelined Datapath



Pipeline principles

- All instructions that share a pipeline must have the same stages in the same order
  - So ADD does nothing during Mem stage
  - sw does nothing during WB stage
- All intermediate values must be latched each cycle
- There is no functional block reuse
  - Eg. We need 2 adders and ALU (like in single cycle)