Chapter 4

Pipelining

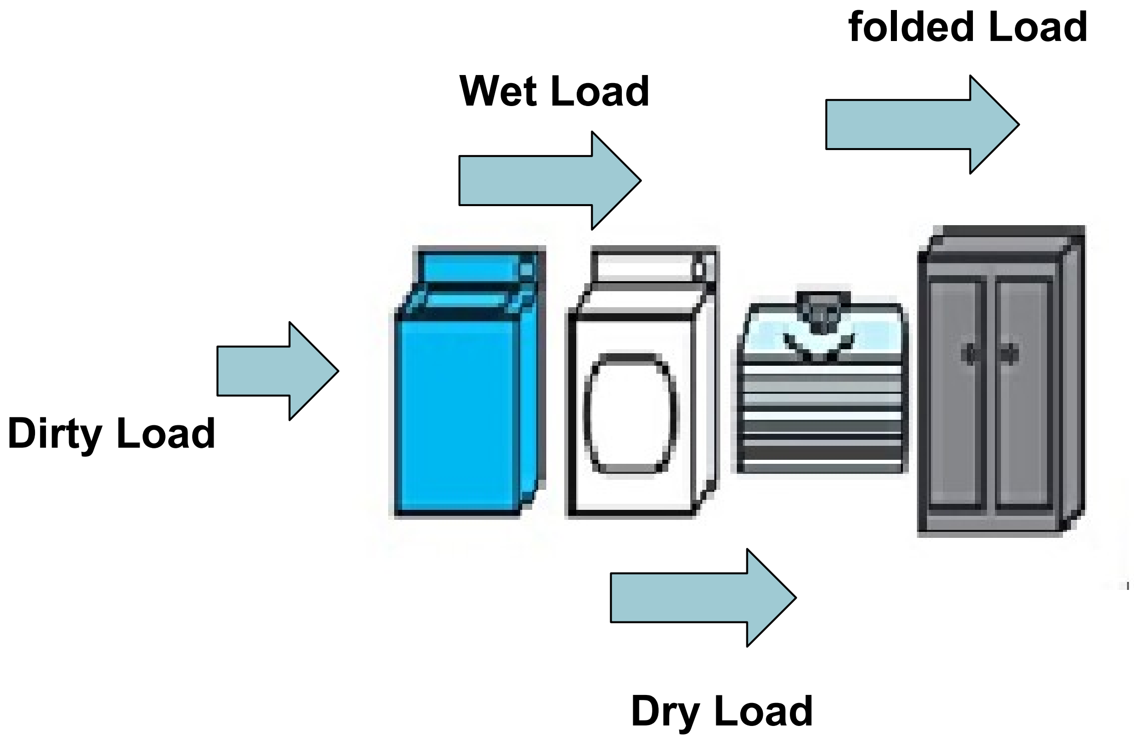
We started with the **single-cycle implementation**, in which a single instruction is executed over a single cycle. In this scheme, a cycle's clock period must be defined to be as long as necessary to execute the longest instruction. But this results in a lot of waste – both in terms of time and space since we need multiple of the same kinds of datapath elements to execute a single instruction.

Then, we looked at **multi-cycle implementation**. In this scheme, instructions are broken up over general steps and each step is performed over a single clock cycle. As a result, we have a smaller clock cycle and we are able to reuse datapath elements in different cycles. However, we are still limited to executing one instruction at a time

pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution
- Today pipelining is key to making processors fast
- People who has done a lot of laundry has intuitively used pipelining

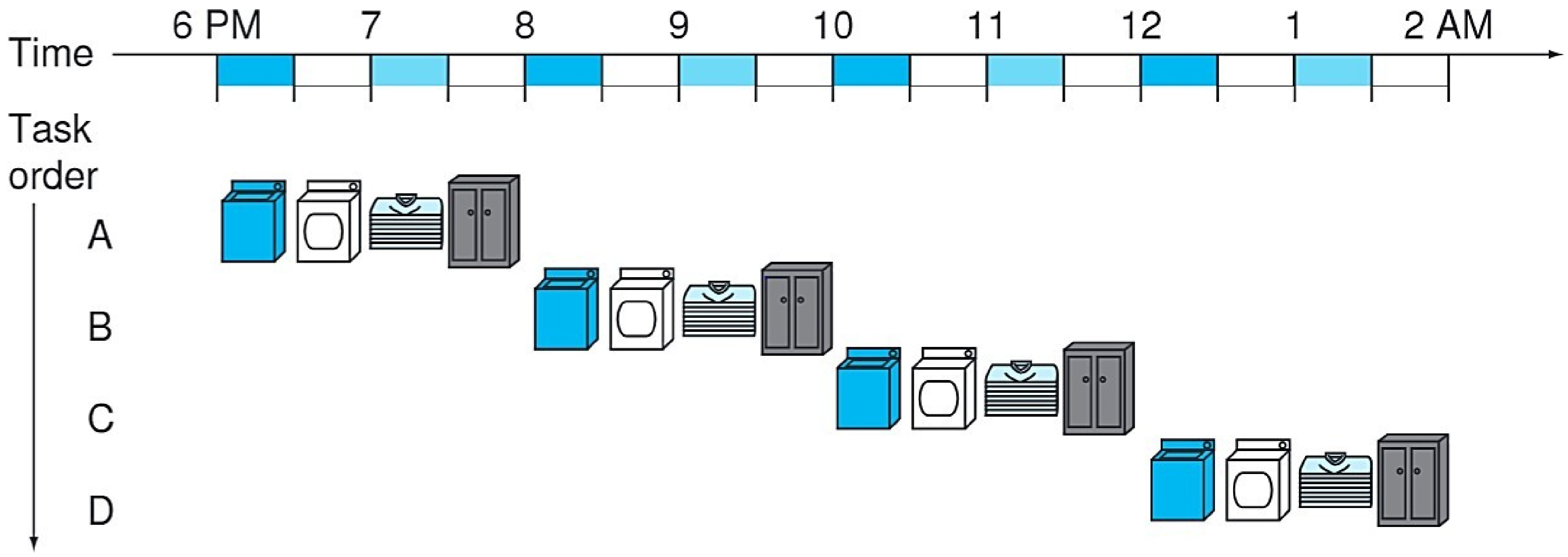
Laundry Analogy



- laundry has following functional units: washing, drying, folding, and putting away.

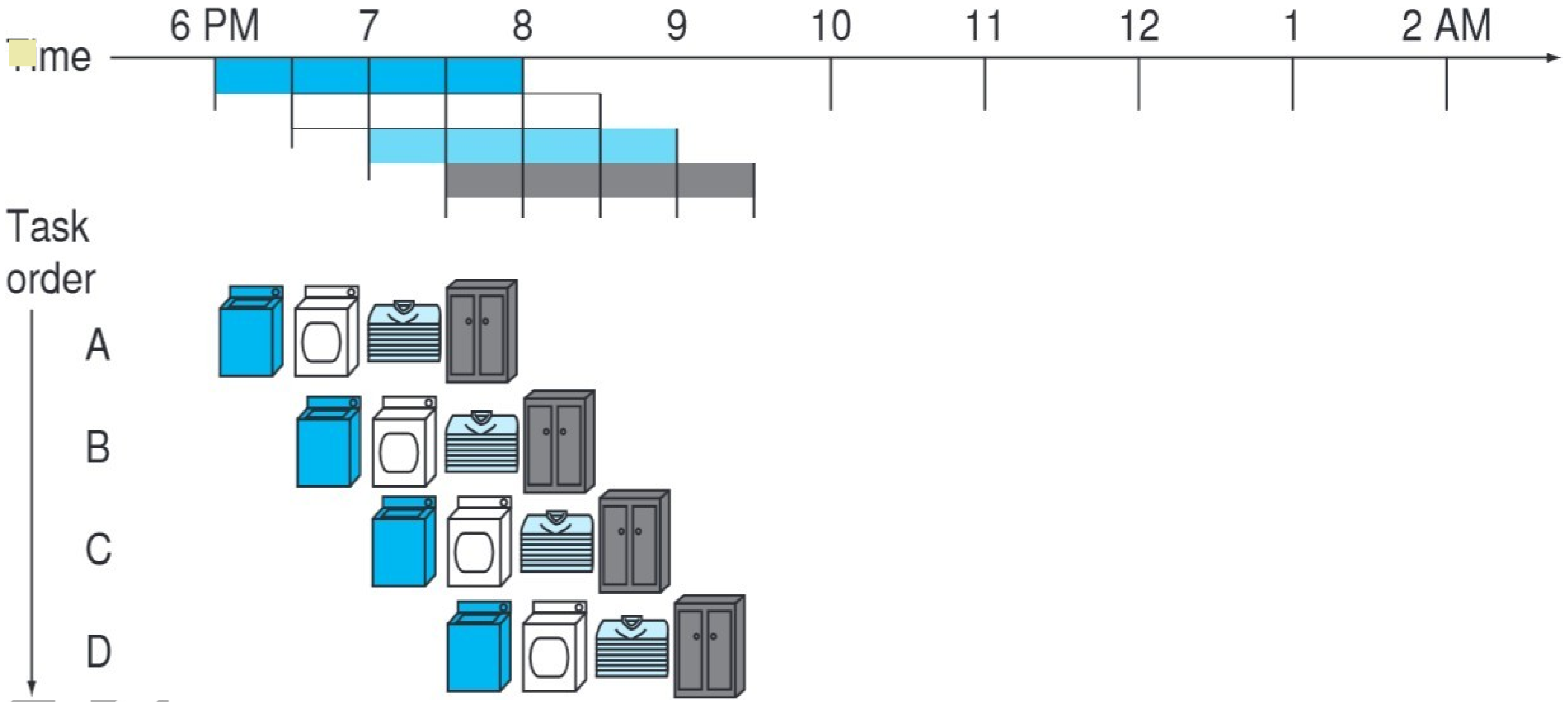
LAUNDRY ANALOGY

- Assume, each step requires thirty minutes and start the next load after the previous load has finished. This is similar to multi-cycle implementation.



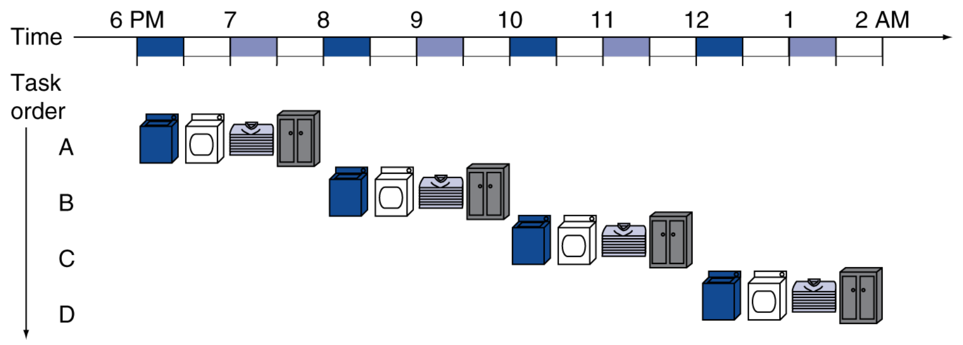
LAUNDRY ANALOGY

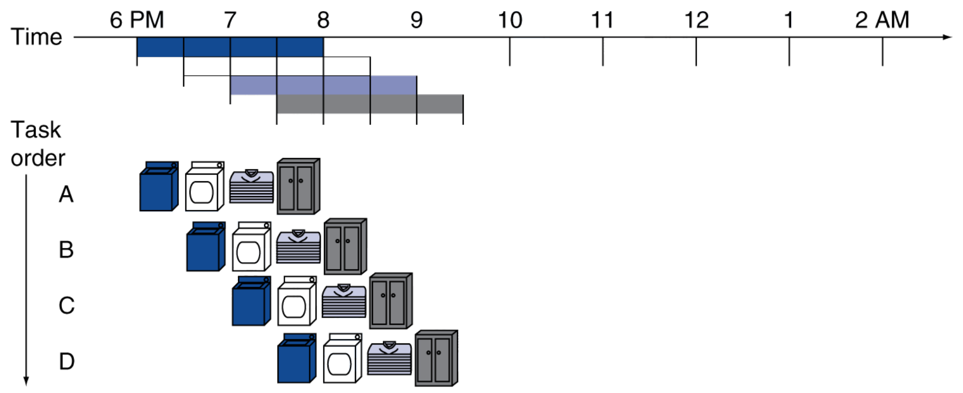
There's no reason why we shouldn't start the next load right after the first load is out of the washer. The washer is available now, after all. This is analogous to pipelining.



Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance





- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

MIPS Pipeline

- Same principles apply to processors where we pipeline instruction execution
- MIPS instruction classically take 5 steps.
- We already know roughly what the stages are:
  - **IF** – Instruction Fetch.
  - **ID** – Instruction Decode.
  - **EX** – Execution or Address Calculation.
  - **Mem** – Data Memory Access.
  - **WB** – Write Back.

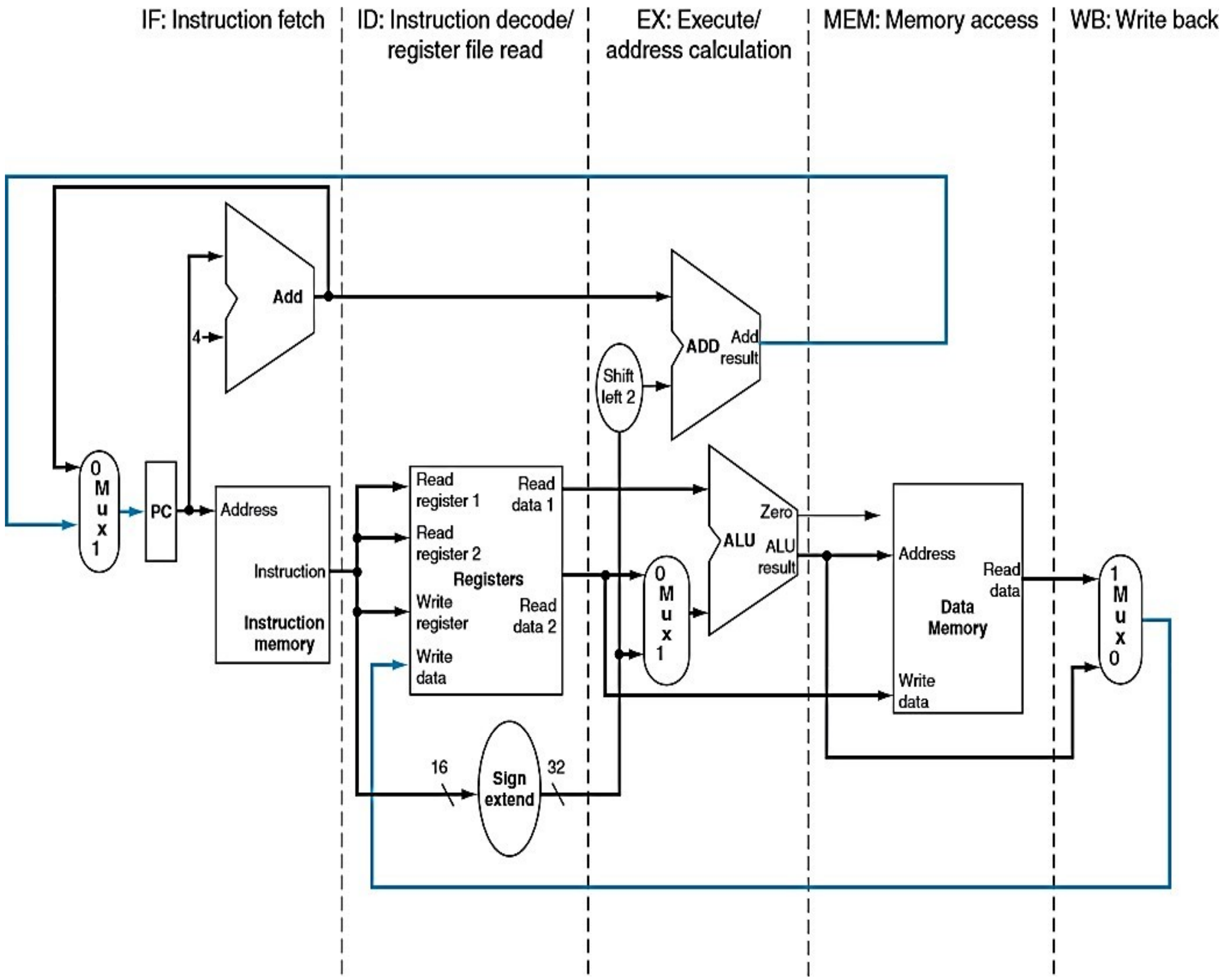
PIPELINING STAGES

- IF stage: fetches the instruction from the instruction cache and increments the PC.
- ID stage: decodes the instruction, reads source registers from register file, sign- extends the immediate value, calculates the branch target address and checks if the branch should be taken.
- EX stage: calculates addresses for accessing memory, performs arithmetic/logical operations on either two register values or a register and an immediate.
- MEM stage: load a value from or store a value into the data cache.
- WB stage: update the register file with the result of an operation or a load.

PIPELINING STAGES

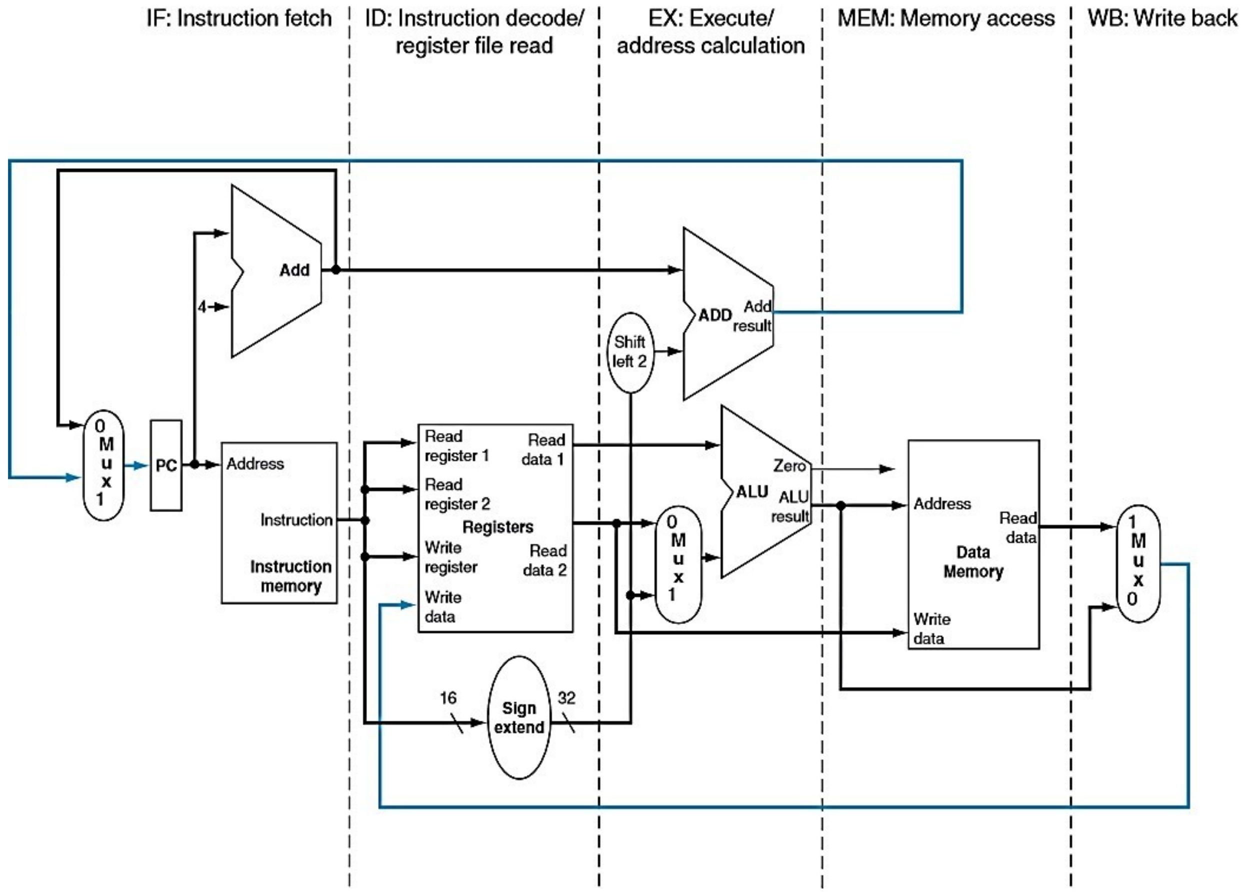
We start by taking the single-cycle datapath and dividing it into 5 stages.

A 5-stage pipeline allows 5 instructions to be executing at once, as long as they are in different stages.

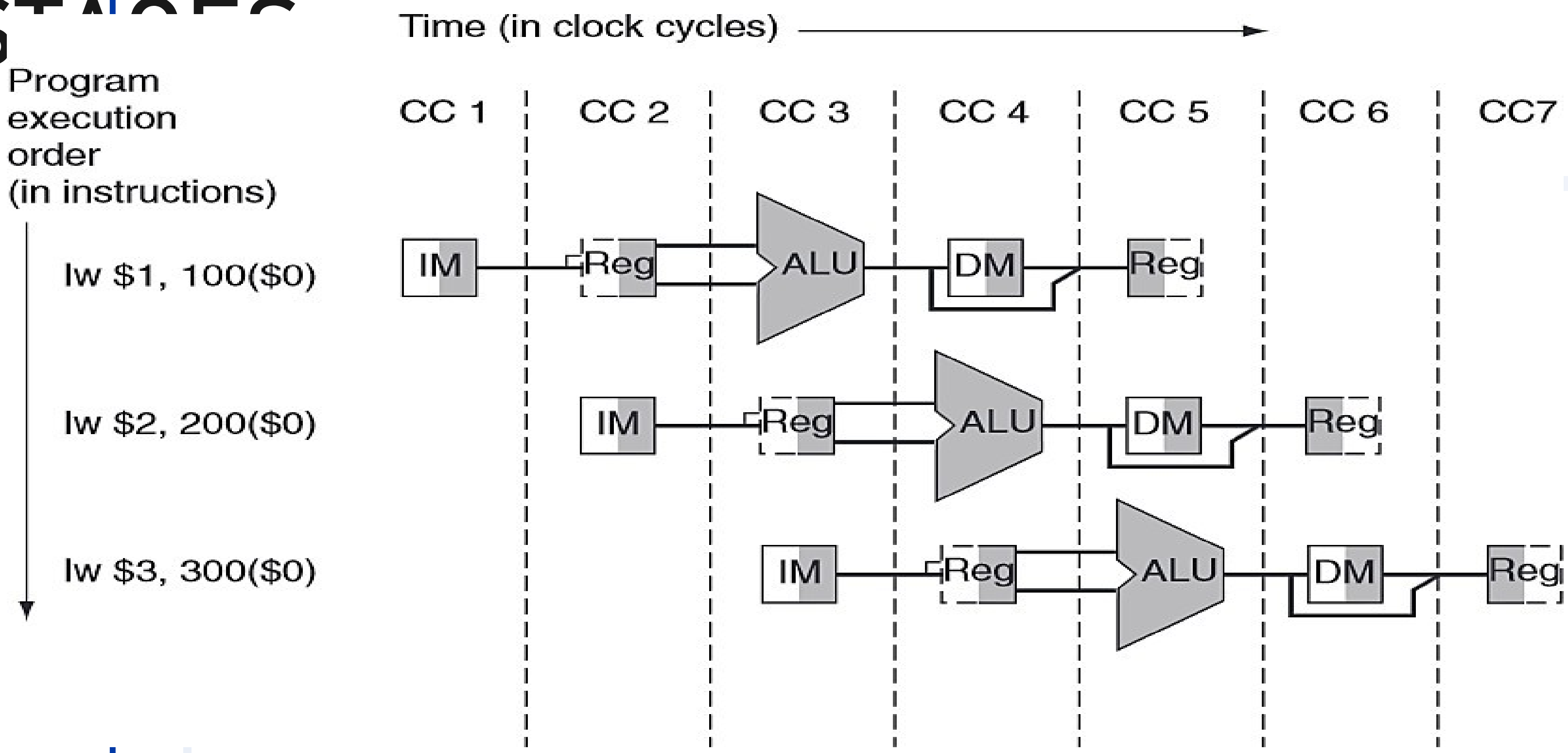


PIPELINING STAGES

All of the data moves from left-to-right with two exceptions: writing to the register file and writing to the PC.



PIPELINING STAGES

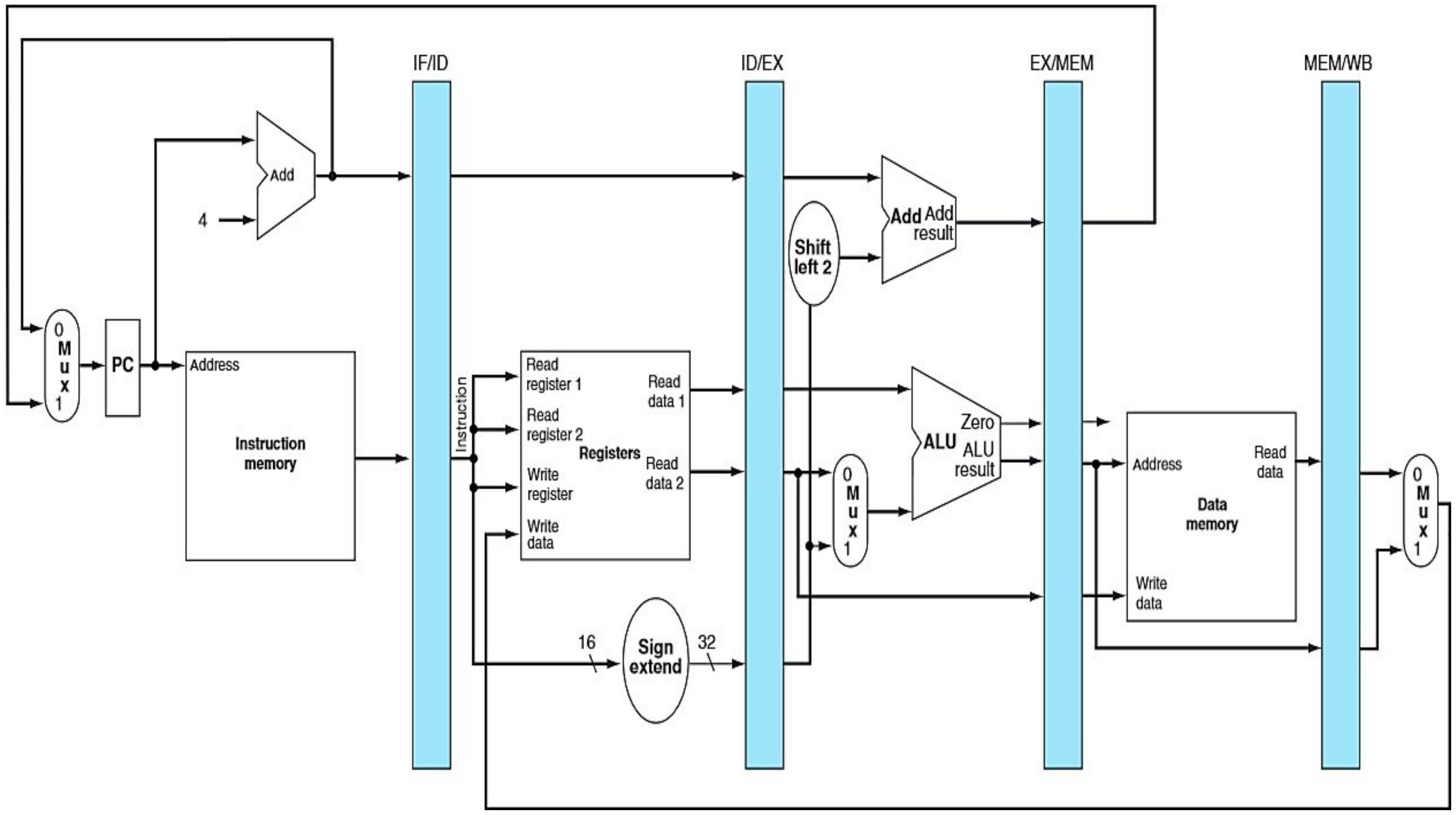


Note that in every cycle, each element is only used by at most one instruction.

PIPELINING STAGES

Even though the datapath is similar to single- cycle, we need to note that we are still executing across multiple cycles.

Therefore, we add **pipeline registers to store data across**

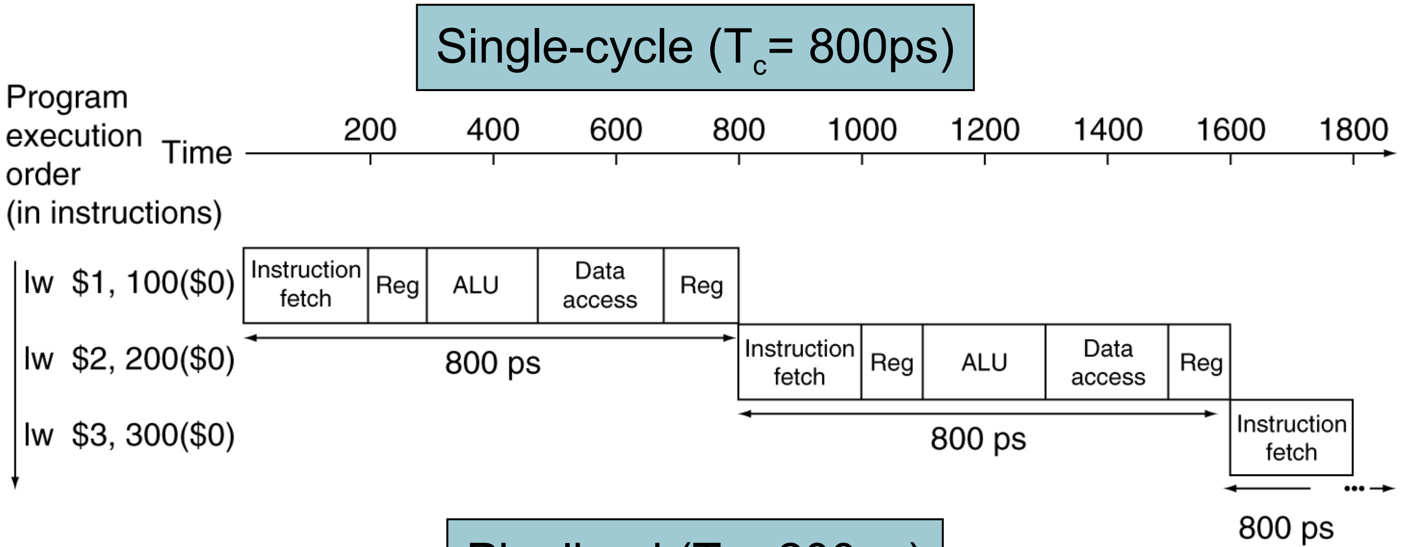


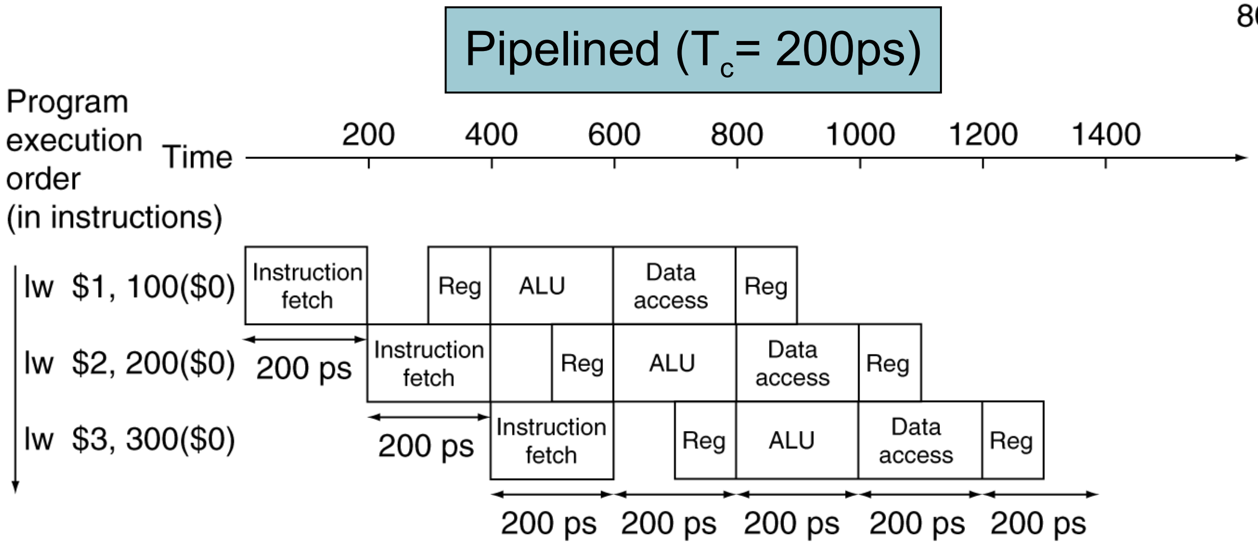
Pipeline Performance

- Let's look at our old example. Say we want to perform three load word instructions in a row. The operation times for the major functional units are 200 ps for memory access, 200 ps for ALU operations, 100 ps for register file read/writes.

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





PIPELINING SPEEDUP

It is important to take note of the fact that the pipelined implementation has a**greater** latency per instruction.

However, this is ok because the advantage we gain with pipelining is increased throughput, which is more important because real programs execute billions of instructions.

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{\text{pipelined}}$

    $$= \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$
  - $\textit{potential}$ speedup = number of pipe stages
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

What happens when we increase no of instructions(1,000,000)

PIPELINING SPEEDUP

As you may have already noticed, our `lw` example does *not* exhibit 5-fold speedup even though there are 5 stages. We have an overall completion time of 2400 ps for single-cycle and an overall completion time of 1400 ps for pipelining. This is merely a 1.7 times speedup.

Imagine instead that we are executing 1,000,000 lw instructions. For single-cycle, this means 800,000,000 ps since each instruction requires 800 ps. But for pipelining, this only means 200,000,800 ps since each additional instruction only adds 200 ps.

$$= \frac{800{,}002{,}400\,ps}{200{,}001{,}400\,ps} \cong \frac{8}{2} = 4$$

When we increase the number of instructions, we get roughly 4 times speedup.

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

Concluding Remarks

- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced