PIPELINE

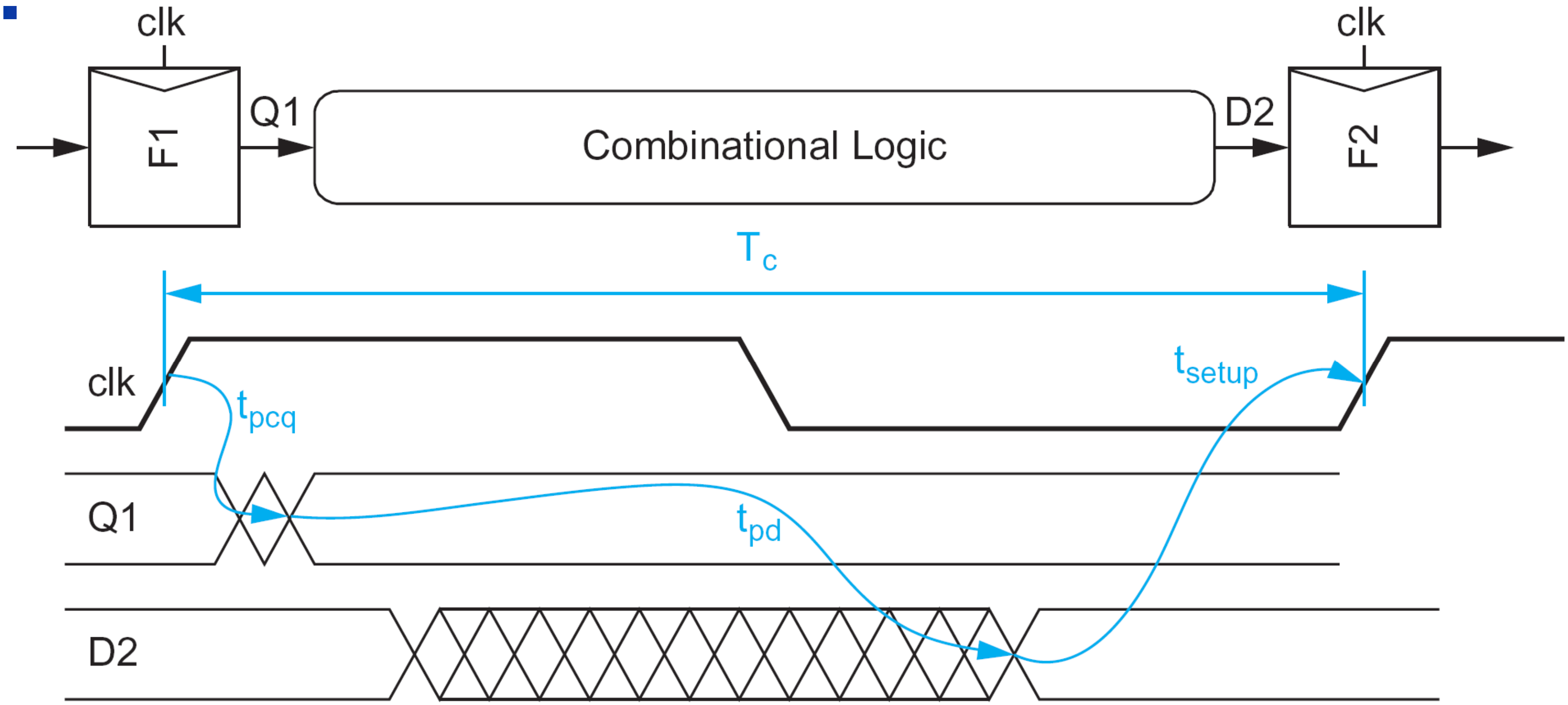
The following material has been adapted from:

Patterson, D. A. and Hennesy, J. L., ***Computer Organization and Design MIPS Edition: The Hardware/Software Interface***, 5th Edition, Morgan Kaufmann

Performance Background

- Clock frequency in ICs is determined by the longest propagation delay.
- Propagation delay is the time it takes for a signal to propagate from
  - An input to a flip-flop
  - A flip-flop to an output
  - A flip-flop to another flip-flop

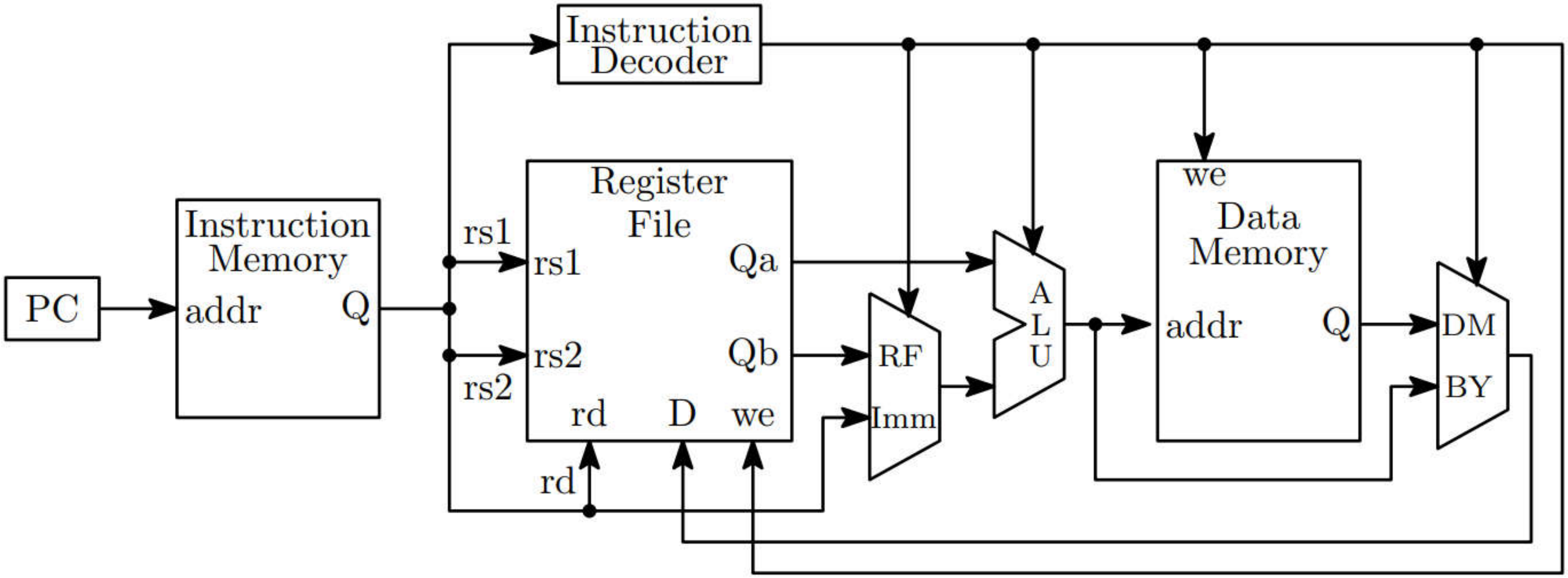
Performance Background



- Clock period should be larger than largest propagation delay
  - Tc > tpd

Performance Background

- Which is the critical path in this simplified MIPS block diagram?



Performance Issues

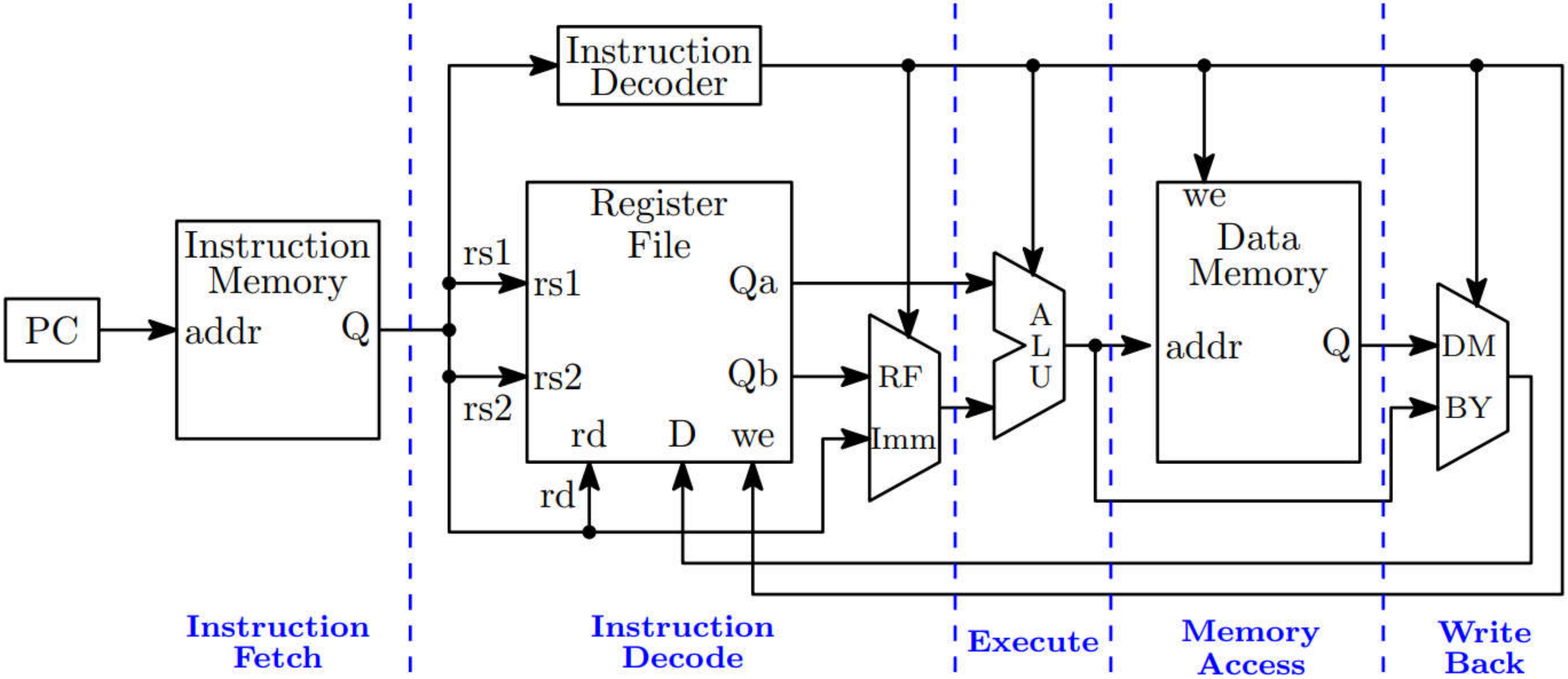
- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- This critical path limits the clock frequency and the number of instructions per second that we could theoretically perform

Performance Issues

- How could we improve our MIPS performance?
  - We could start by splitting the critical path into smaller paths.

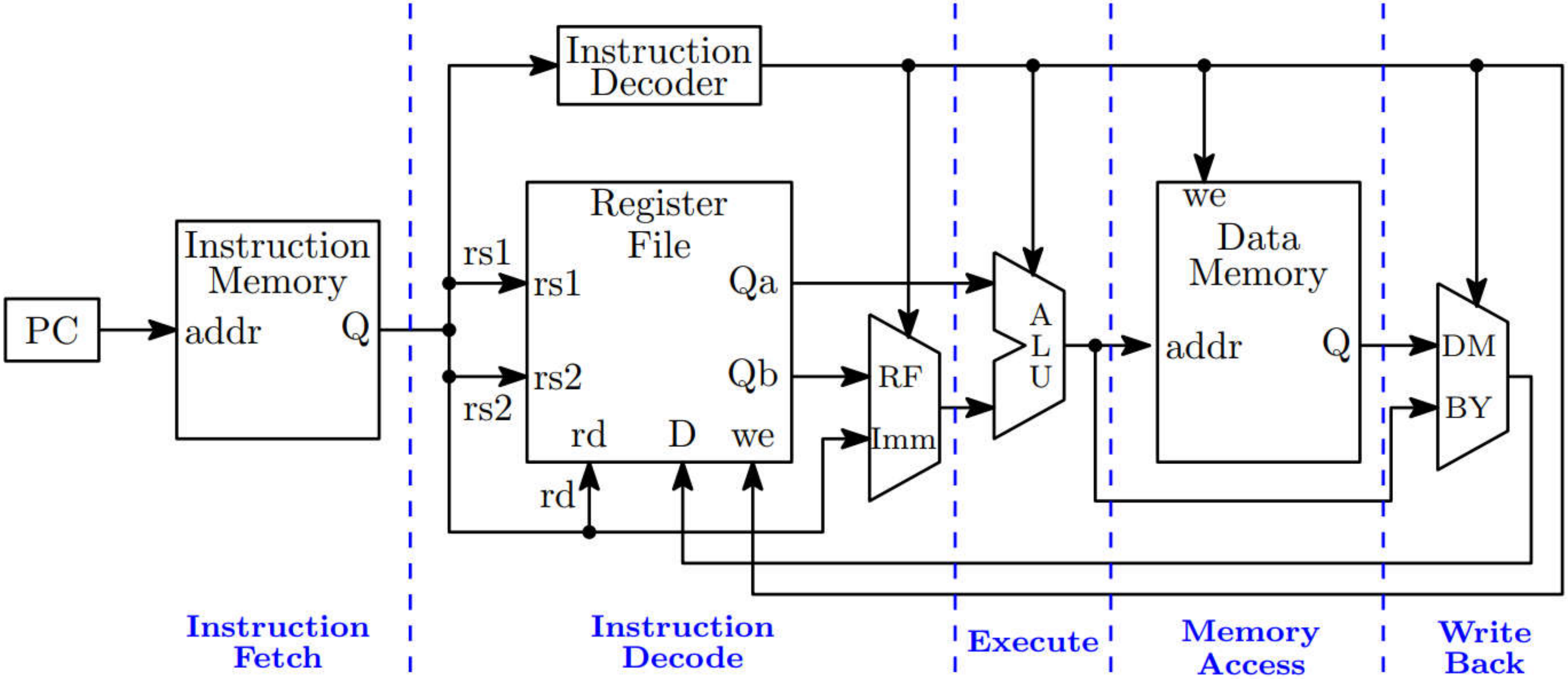
Performance Issues

- We could split our single-cycle execution into different stages.



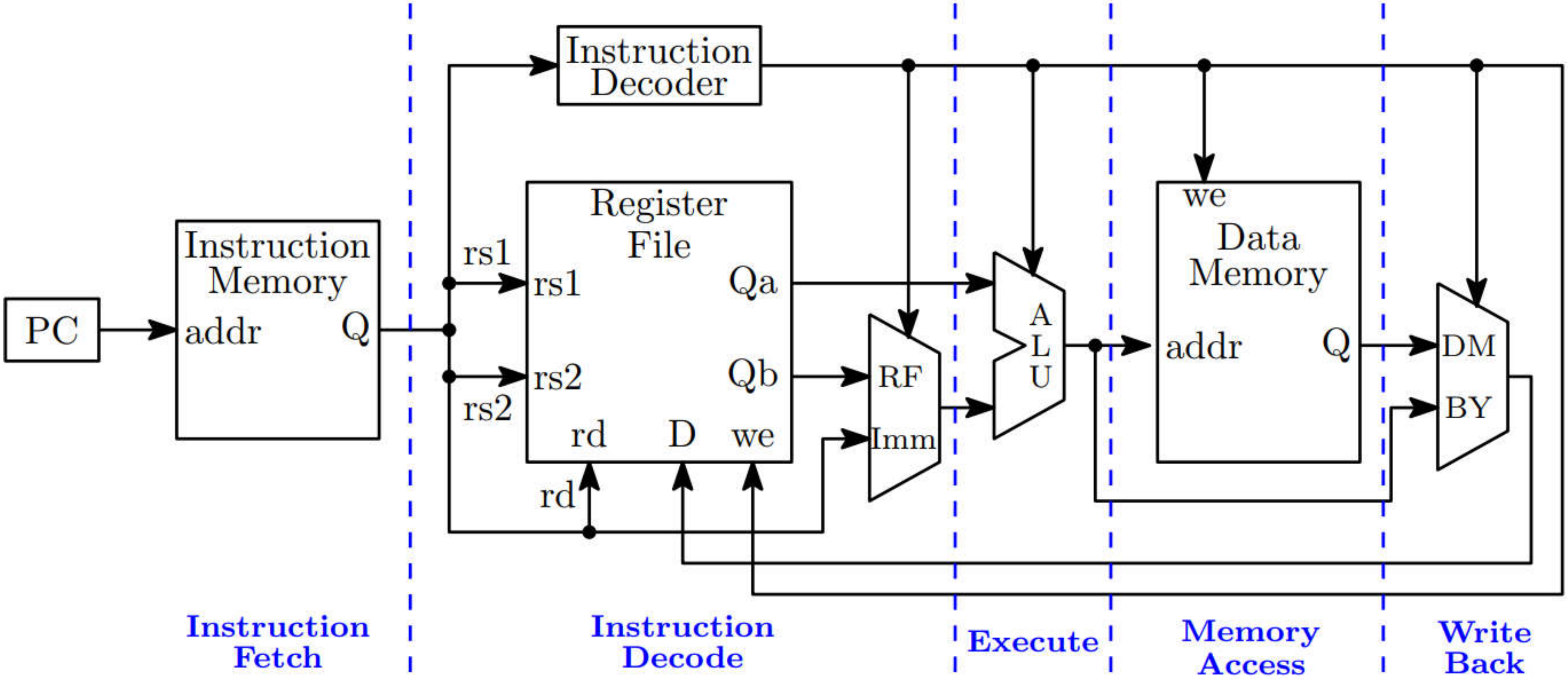
Performance Issues

- This technique is called Pipelining



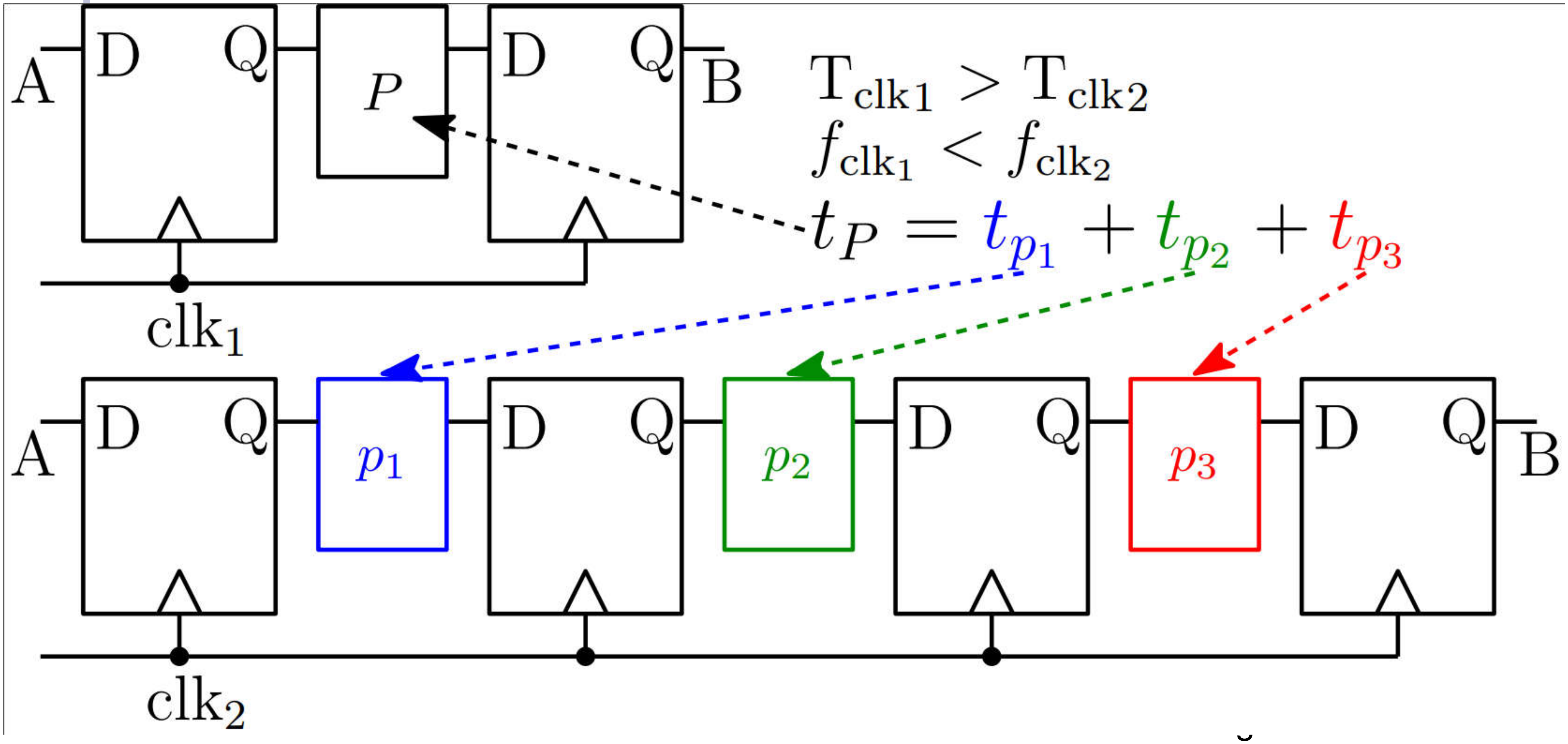
Performance Issues

- This technique is called Pipelining



Pipeline

- Pipelining consist in breaking down a single task into several stages.



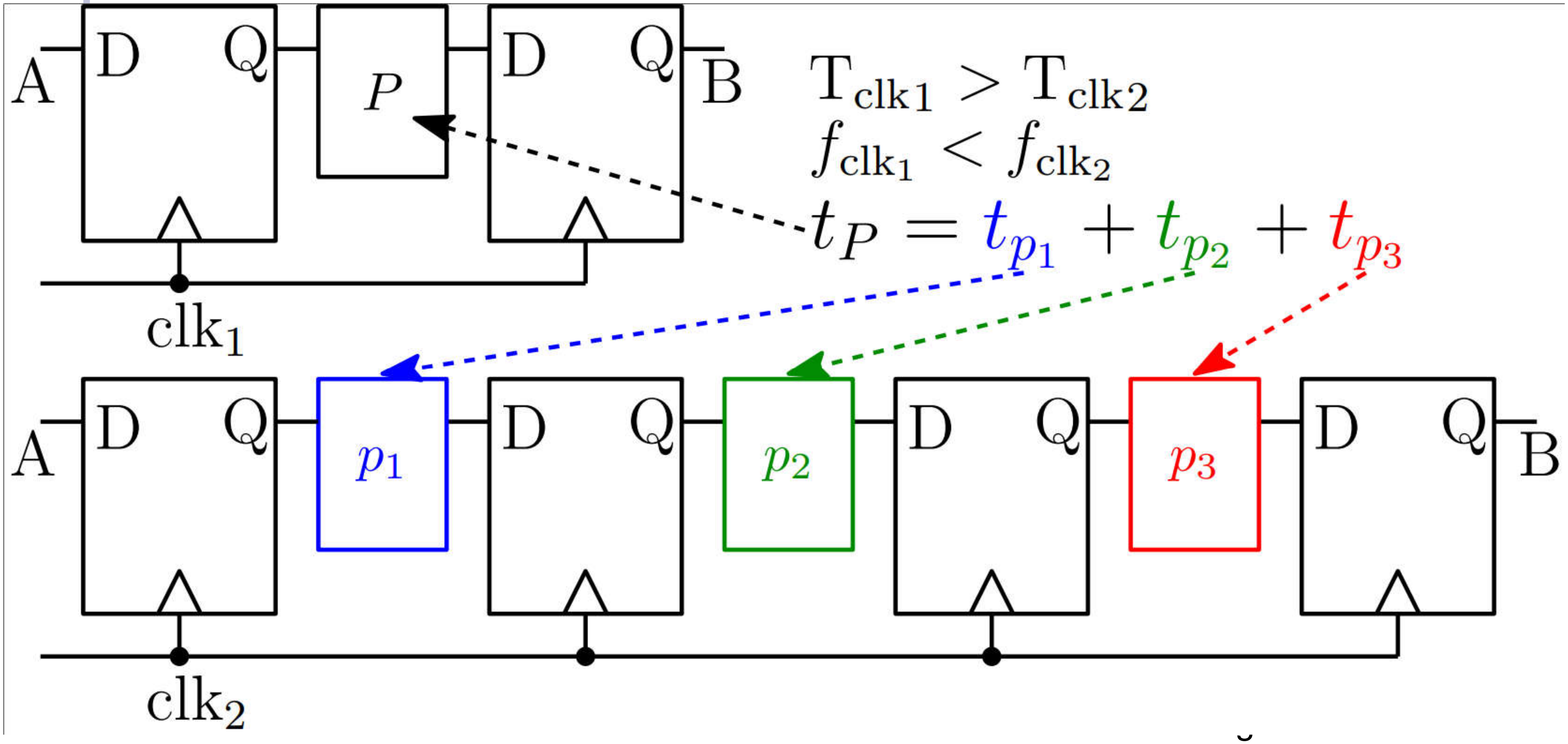
$\mathrm{T_{clk1}} > \mathrm{T_{clk2}}$

$f_{\mathrm{clk}_1} < f_{\mathrm{clk}_2}$

$t_P = t_{p_1} + t_{p_2} + t_{p_3}$

Pipeline

- Task from A to B takes the same time in both cases. What's the advantage?



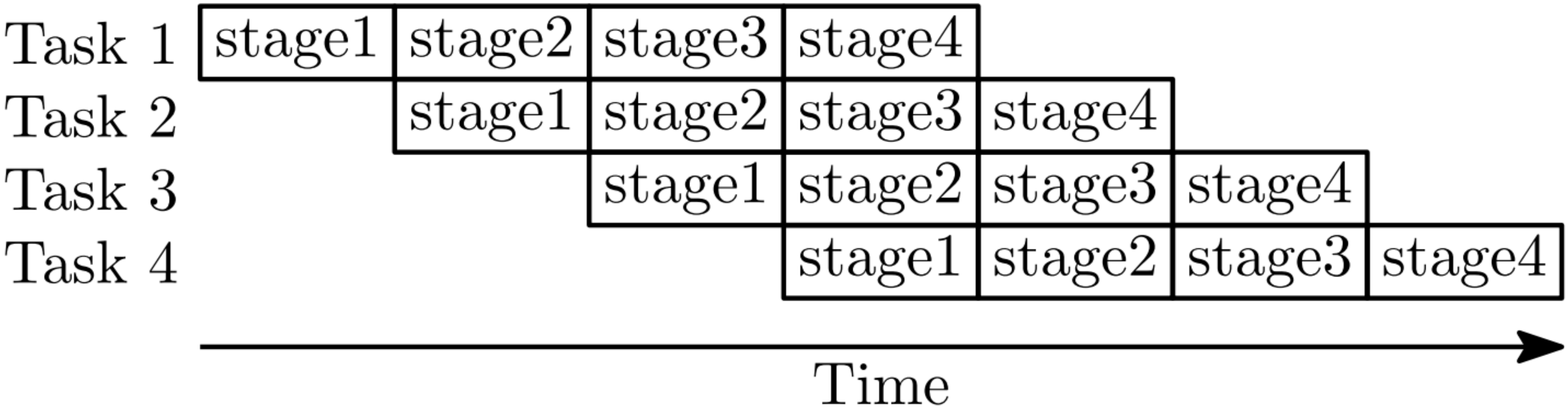
$$T_{clk1} > T_{clk2}$$

$$f_{clk_1} < f_{clk_2}$$

$$t_P = t_{p_1} + t_{p_2} + t_{p_3}$$

Pipeline

- We can perform tasks in parallel



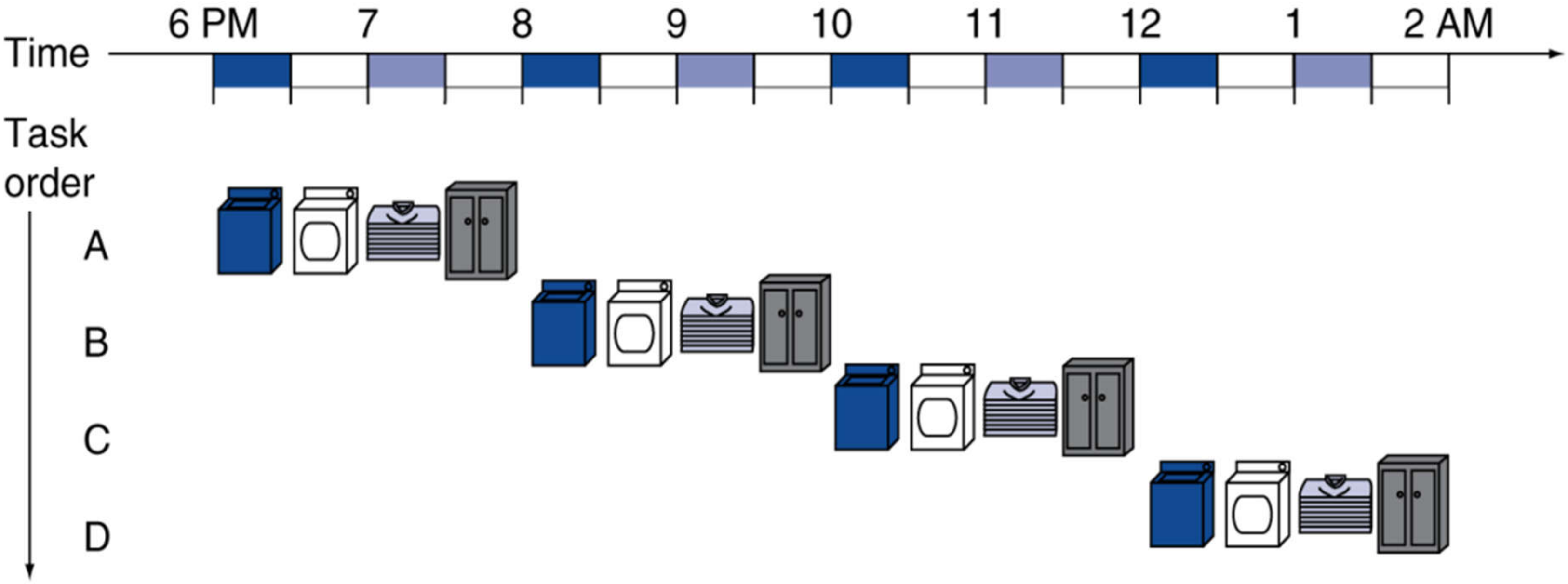
Task 1	stage1	stage2	stage3	stage4			
Task 2		stage1	stage2	stage3	stage4		
Task 3			stage1	stage2	stage3	stage4	
Task 4				stage1	stage2	stage3	stage4

Pipelining Analogy

- Assume you work in a laundry shop.
- You divide your job into the following stages.
  - Wash
  - Dry
  - Fold
  - Store
- Assume each stage takes 30 minutes to complete

Pipelining Analogy

- How long would it take you to complete a single laundry request?
- 30min x 4 stages = 2 hours



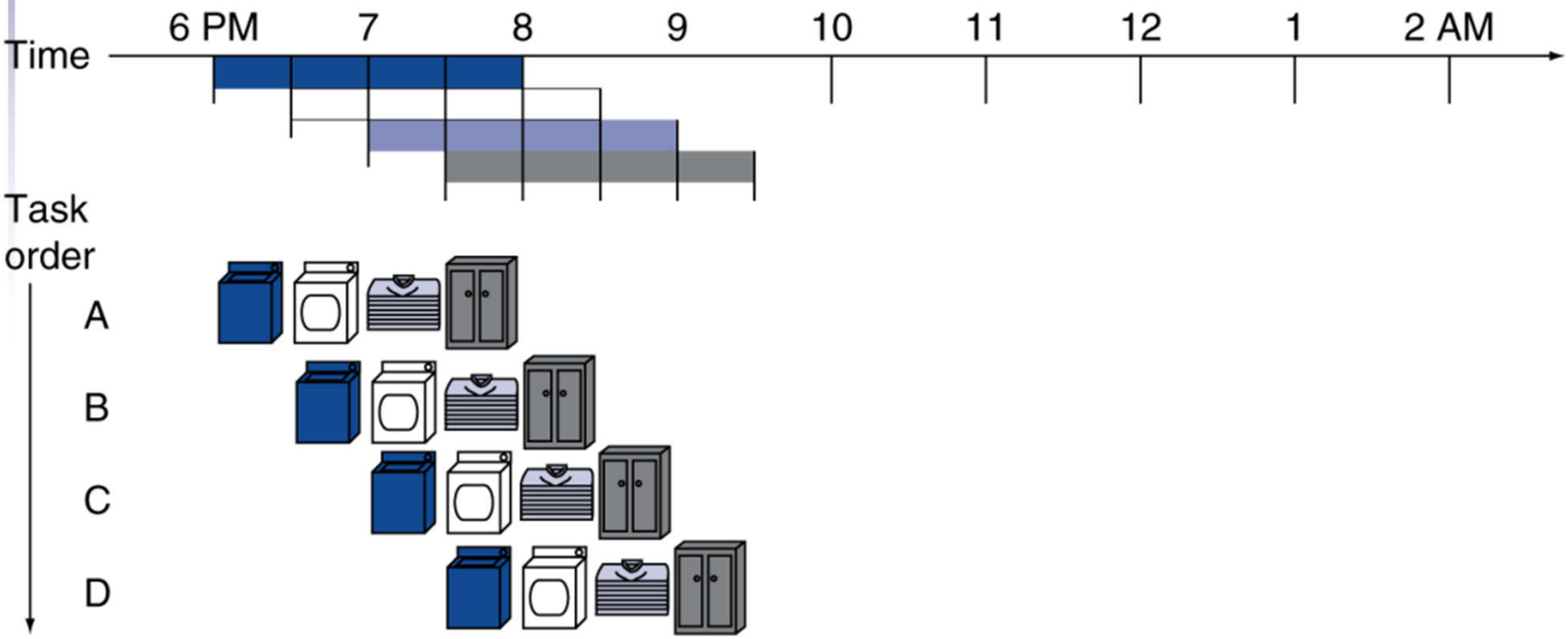
- What about 4 laundry request? – 8 hours

Pipelining Analogy

- Start a new load right after the washing machine becomes available.
- Dry first load and wash second load at the same time.
- Fold first load, dry second load and wash third load at the same time
- Continue with this principle until you complete all four loads

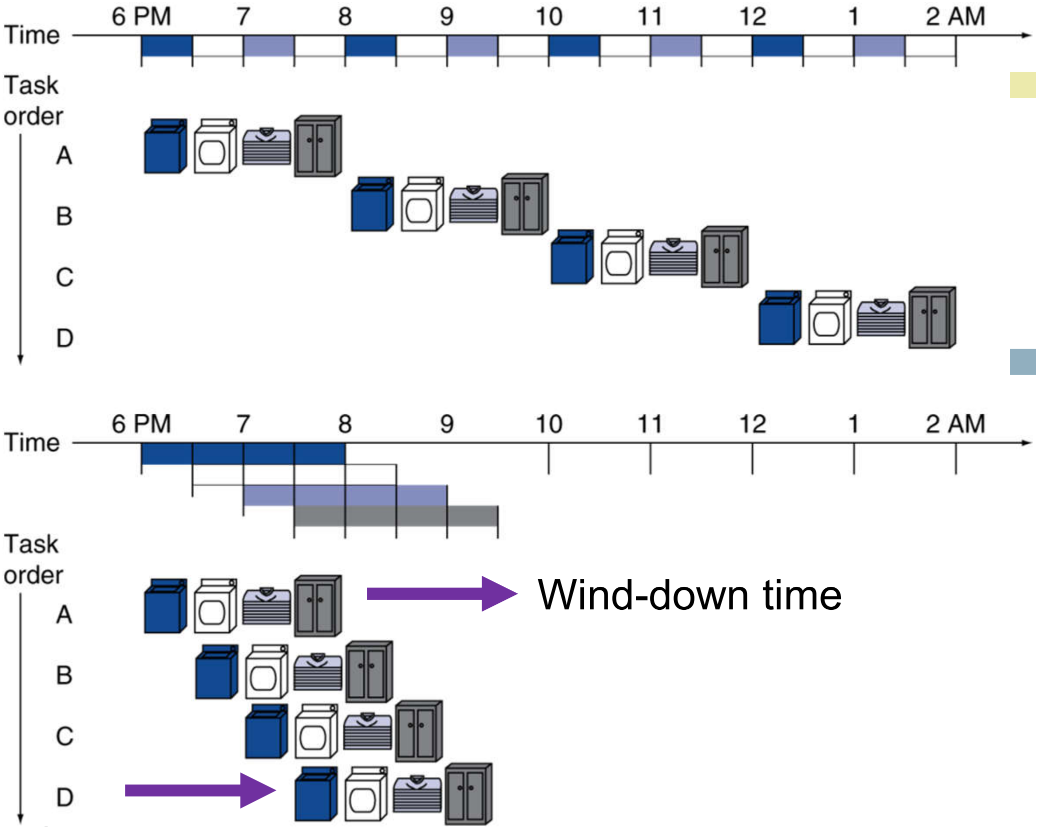
Pipelining Analogy

- 4 loads now takes 3.5 hours



Pipelining Analogy

- Improved performance



Wind-down time

Start-up time

Full pipeline: All resources are used at the same time

- Four loads:
  - Speedup = 8/3.5 = 2.3
- What about infinite number of loads?

Pipelining Speedup Equation

m is the number of tasks

n is the number of stages per task

t is the time required to complete a stage

$$T_{seq} = m \cdot \sum_{i=1}^{n} t_i$$

If all n stages take the same time t, then

$$T_{seq} = m \cdot n \cdot t$$

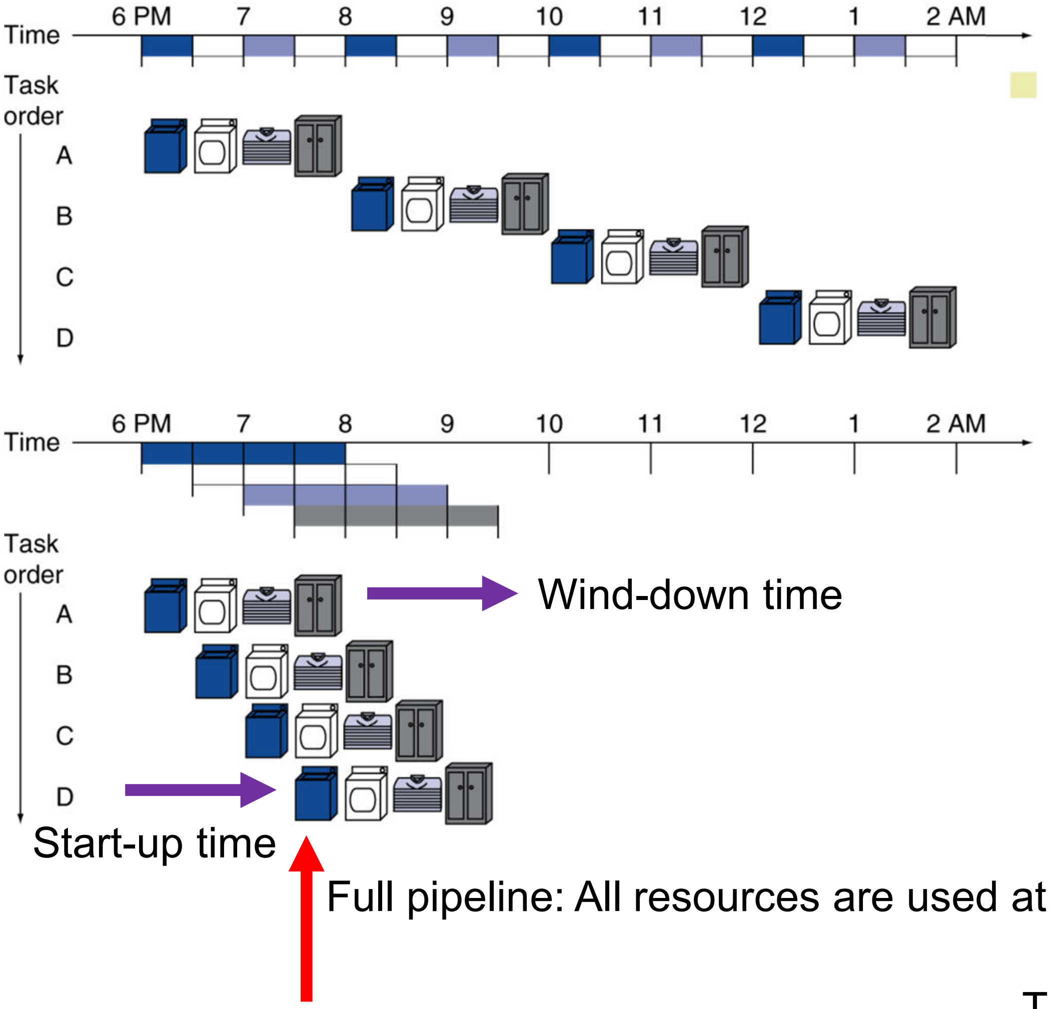
$$T_{pipe} = (m - 1) \cdot t + n \cdot t$$

$$S = \frac{T_{seq}}{T_{pipe}} = \frac{m \cdot n \cdot t}{(m-1) \cdot t + n \cdot t} = \frac{m \cdot n}{(m-1)+n}$$

$$\lim_{m \to \infty} \frac{m \cdot n}{(m-1)+n} = n$$

Pipelining Speedup

- For the laundry analogy



Wind-down time

Start-up time

Full pipeline: All resources are used at the same time

- Infinite loads:
  - Speedup ≈ 4

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

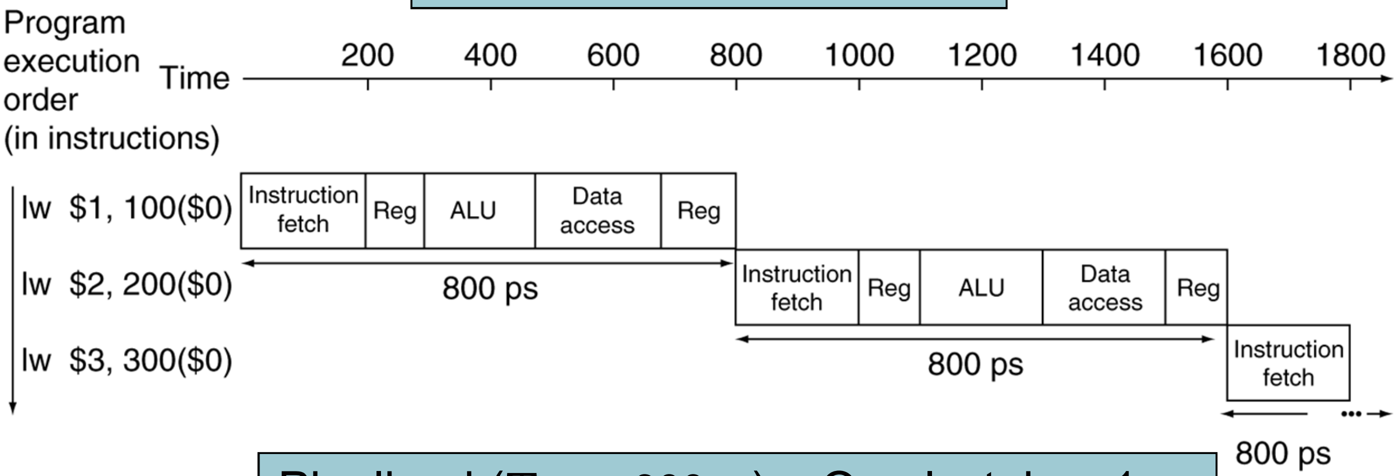
Pipeline Performance

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

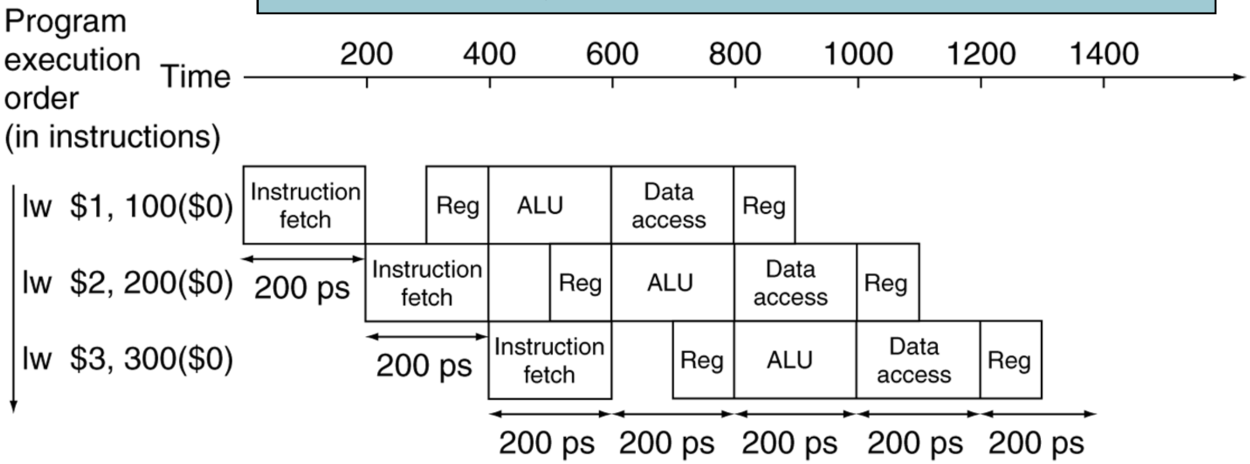
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_{clk}= 800ps)



Pipelined (T_{clk}= 200ps) – One lw takes 1ns



Pipeline Speedup

- If all stages are balanced
  - i.e., all stages take the same time
  - $\text{Time between instructions}_{\text{pipelined}} = \dfrac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to **increased throughput**
  - Latency per instruction does not decrease

Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

PIPELINE HAZARDS

Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structural hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

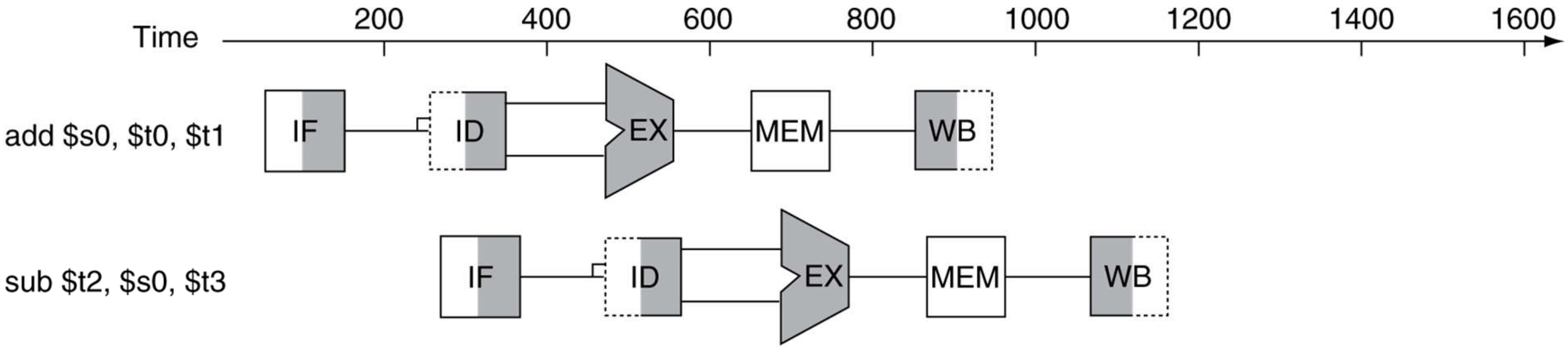
Structural Hazards

- HW cannot support the combination of instructions in the same clock cycle.
- Conflict for use of a resource.
- In MIPS pipeline with a **single memory.**
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Data Hazards

- An instruction depends on completion of data access by a previous instruction

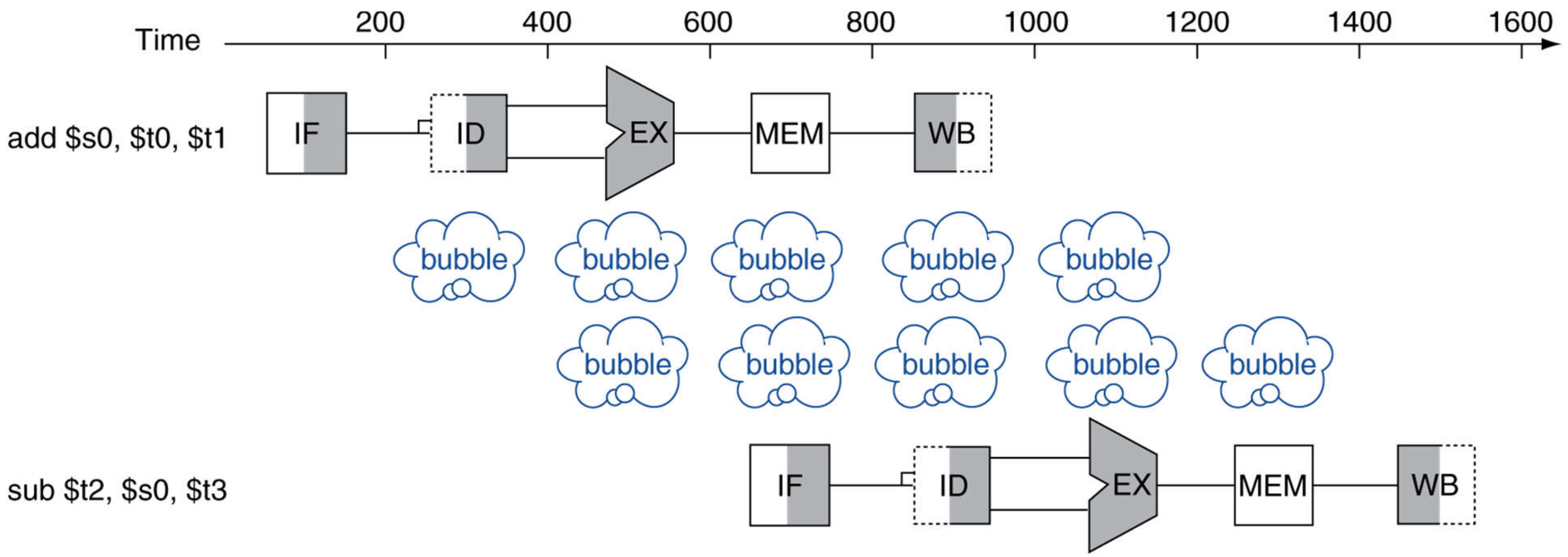
  -
    ```
    add  $s0, $t0, $t1
    sub  $t2, $s0, $t3
    ```



Data Hazards

- An instruction depends on completion of data access by a previous instruction

  -
    ```
    add  $s0, $t0, $t1
    sub  $t2, $s0, $t3
    ```

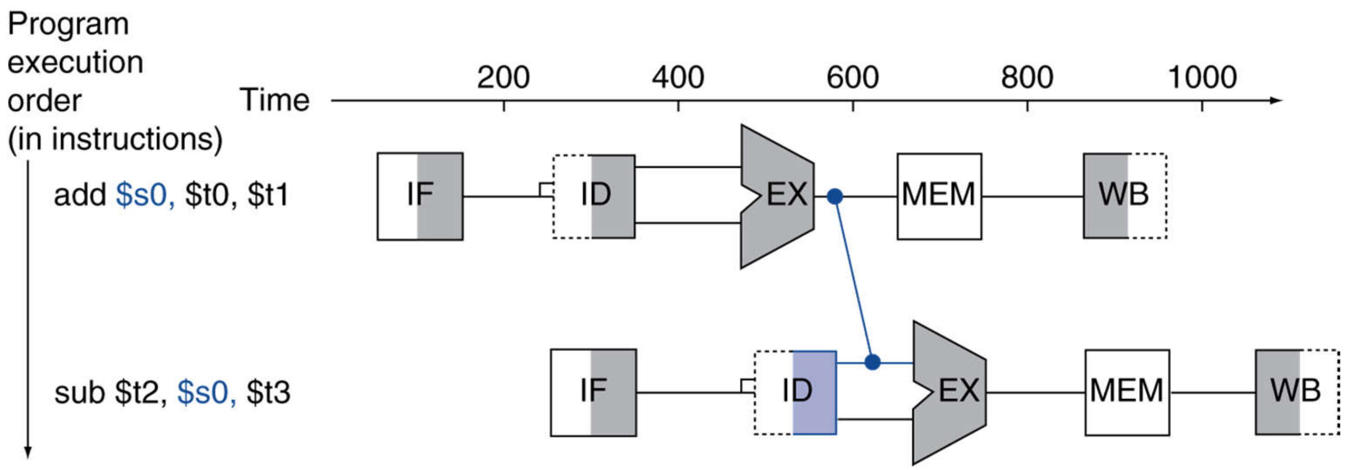


How can we avoid pipeline bubbles?

- We can rely on the compiler
  - Difficult task and extended compile times
- Data forwarding (bypassing)
  - As soon as the data is computed, it can be used by the next instruction.
  - This is not a hazard-free solution. There may be pipeline bubbles due to load instructions.

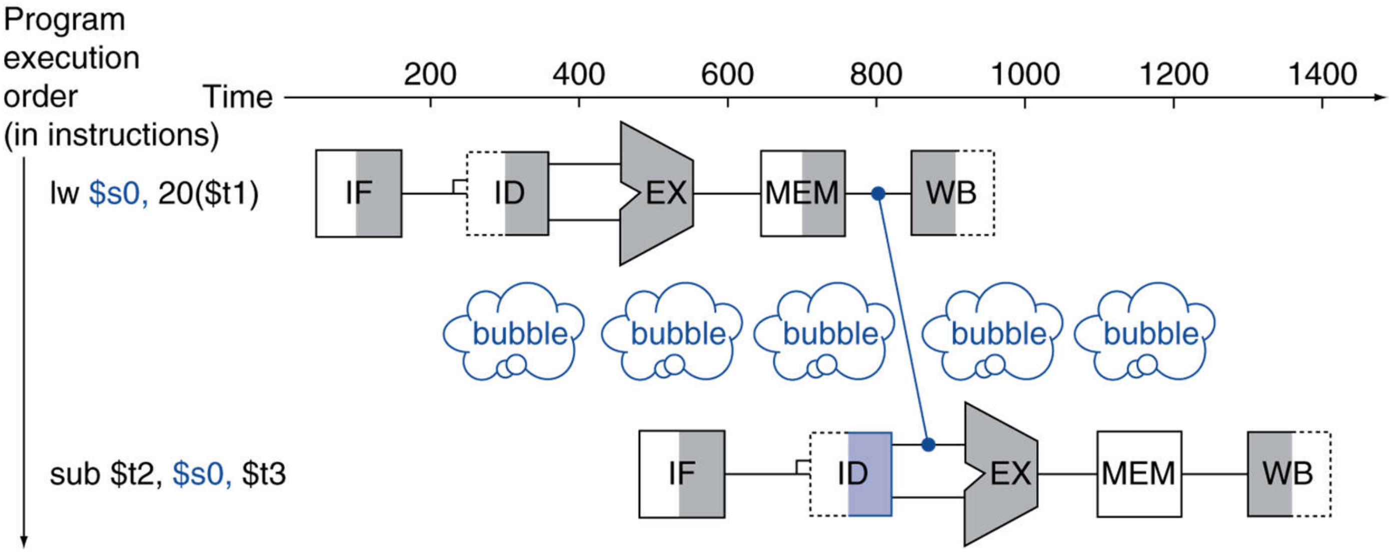
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don’t wait for it to be stored in a register
  - Requires extra connections in the datapath



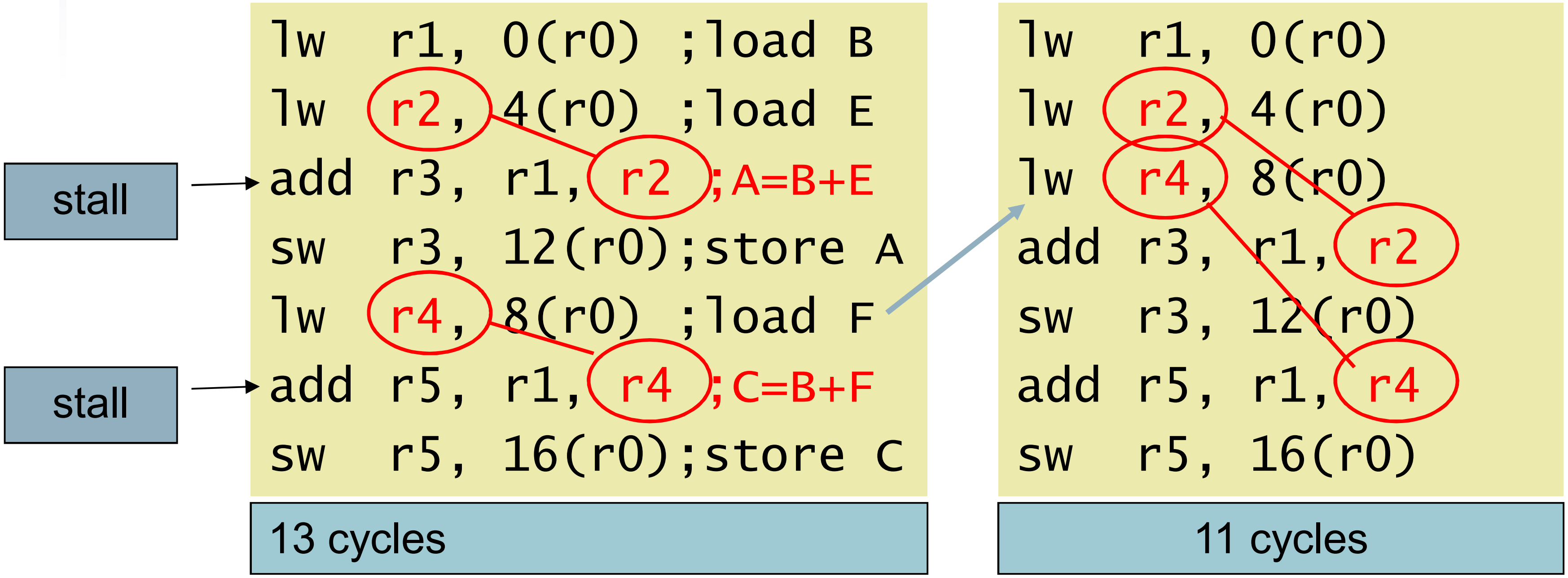
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!
- Pipeline stall (bubble)



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- A = B + E; C = B + F;
- B=Mem[0]; E=Mem[4]; F=Mem[8]



```
lw   r1, 0(r0) ;load B
lw   r2, 4(r0) ;load E
add  r3, r1, r2 ;A=B+E
sw   r3, 12(r0);store A
lw   r4, 8(r0) ;load F
add  r5, r1, r4 ;C=B+F
sw   r5, 16(r0);store C
```

13 cycles

```
lw   r1, 0(r0)
lw   r2, 4(r0)
lw   r4, 8(r0)
add  r3, r1, r2
sw   r3, 12(r0)
add  r5, r1, r4
sw   r5, 16(r0)
```

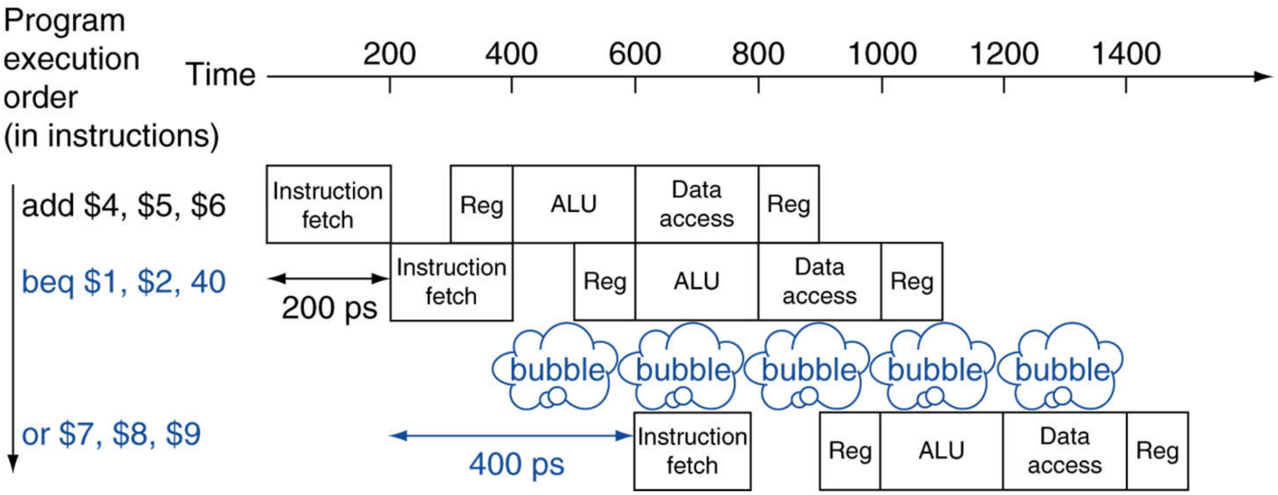
11 cycles

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

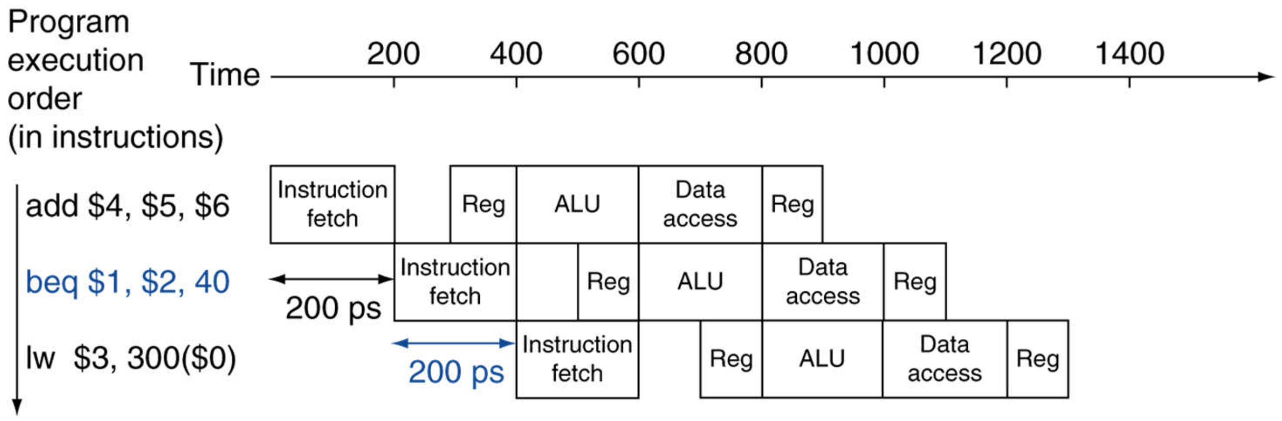


Branch Prediction

- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct


Program execution order (in instructions)

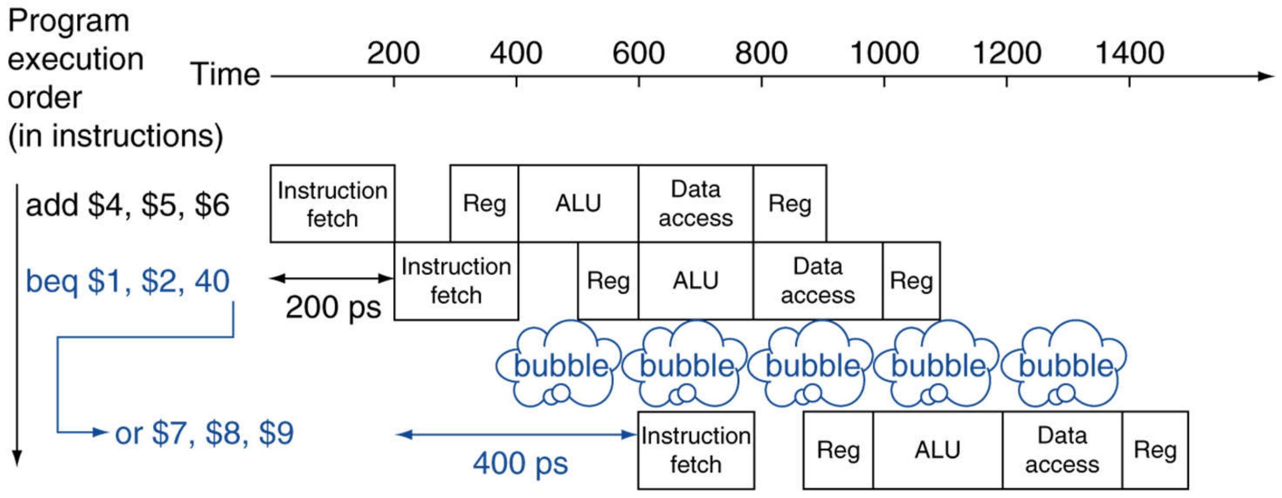
Time 200 400 600 800 1000 1200 1400

add $4, $5, $6 — Instruction fetch | Reg | ALU | Data access | Reg

beq $1, $2, 40 — 200 ps — Instruction fetch | Reg | ALU | Data access | Reg

lw $3, 300($0) — 200 ps — Instruction fetch | Reg | ALU | Data access | Reg

Prediction incorrect


Program execution order (in instructions)

Time 200 400 600 800 1000 1200 1400

add $4, $5, $6 — Instruction fetch | Reg | ALU | Data access | Reg

beq $1, $2, 40 — 200 ps — Instruction fetch | Reg | ALU | Data access | Reg

bubble bubble bubble bubble bubble

or $7, $8, $9 — 400 ps — Instruction fetch | Reg | ALU | Data access | Reg

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation