CS429: Computer Organization and Architecture

Pipeline I

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: July 11, 2019 at 09:17

Overview

What's wrong with the sequential (SEQ) Y86?

- It's slow!
- Each piece of hardware is used only a small fraction of the time.
- We would like to find a way to get more performance with only a little more hardware.

General Principles of Pipelining

- Express task as a collection of stages
- Move instructions through stages
- Process several instructions at any given moment

Overview

Creating a Pipelined Y86 Processor

- Rearrange SEQ
- Insert pipeline registers
- Deal with data and control hazards

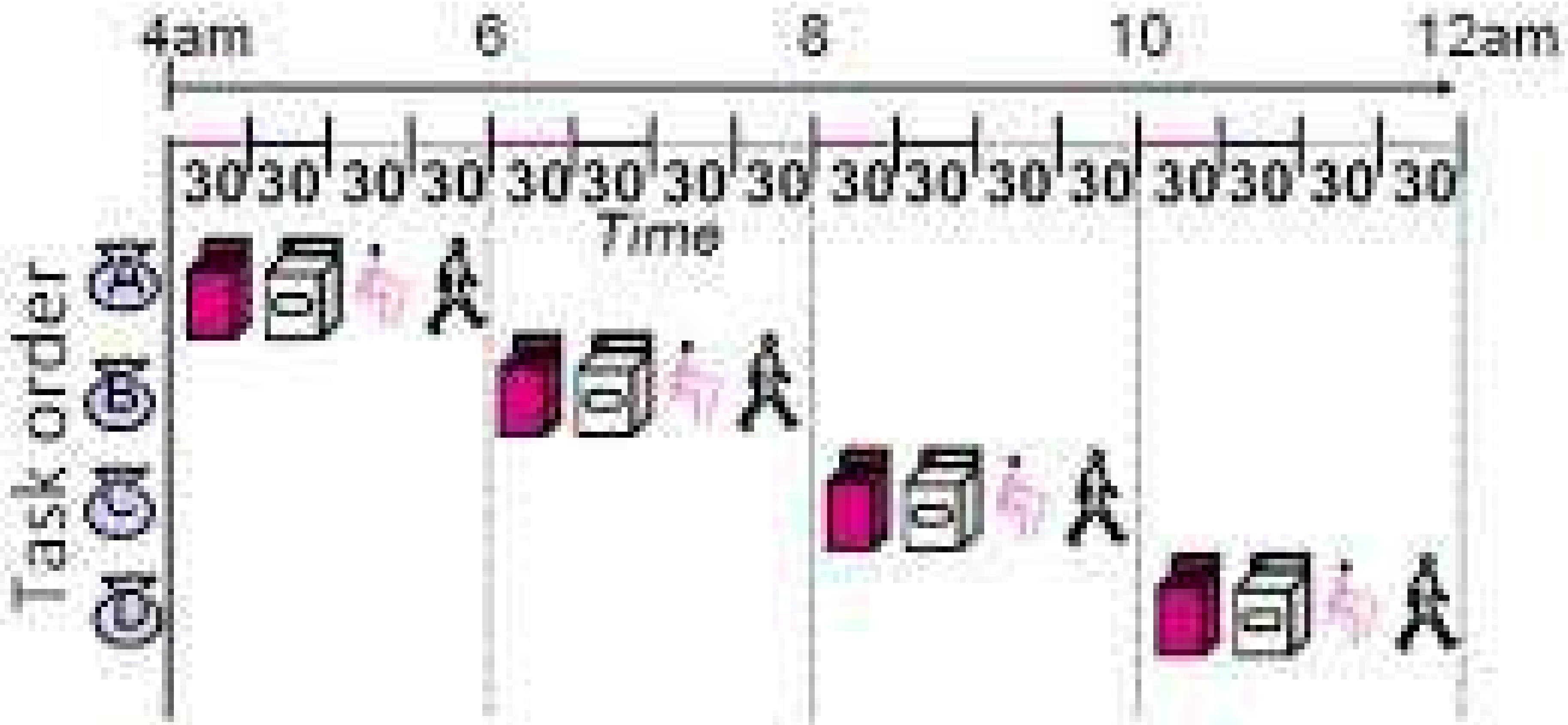
Pipelining is an optimization to the implementation. Like any other optimization, it should not change the semantics.

Pipeline Correctness Axiom: A pipeline is correct only if the resulting machine satisfies the ISA (nonpipelined) semantics.

Suppose you have four folks, each with a load of clothes to wash, dry, fold and stash away. There are four subtasks: wash, dry, fold, stash. Suppose each takes 30 minutes.

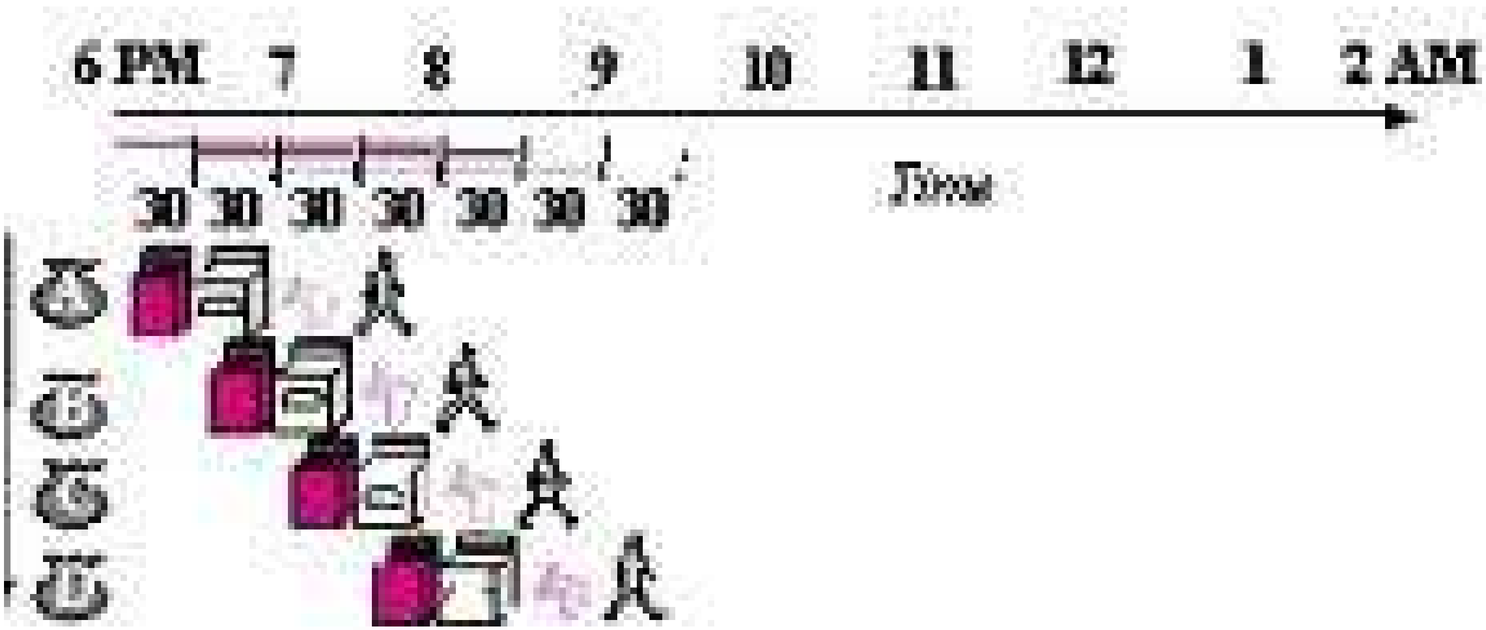
Time to do a load of laundry from start to finish: 2 hours. (That's the *latency*.)

Sequential Laundry



- Sequential laundry takes 8 hours for 4 loads.
- What would it mean to pipeline this process?
- If they learned pipelining, how long would 4 loads take?

Pipelined Laundry



Pipelined laundry takes 3.5 hours for 4 loads! But each load still takes 2 hours.

What's the metric that improved? How would you measure the efficiency of the process if you were running a laundry service with loads (inputs) always ready to process?

Latency vs. Throughput

Latency is the time from start to finish for a given task.

Throughput is the number of tasks completed in a given time period.

Example: suppose that each laundry stage (wash, dry, fold, stash) takes 30 minutes. But you have a laundromat with 4 washers, 4 driers, 4 folding stations, 4 stashing stations.

- What is the latency?

Latency vs. Throughput

Latency is the time from start to finish for a given task.

Throughput is the number of tasks completed in a given time period.

Example: suppose that each laundry stage (wash, dry, fold, stash) takes 30 minutes. But you have a laundromat with 4 washers, 4 driers, 4 folding stations, 4 stashing stations.

- What is the latency?
  Latency is 2 hours, because it still takes 2 hours to get any single load through the entire process.
- What is the highest possible throughput (per hour)?

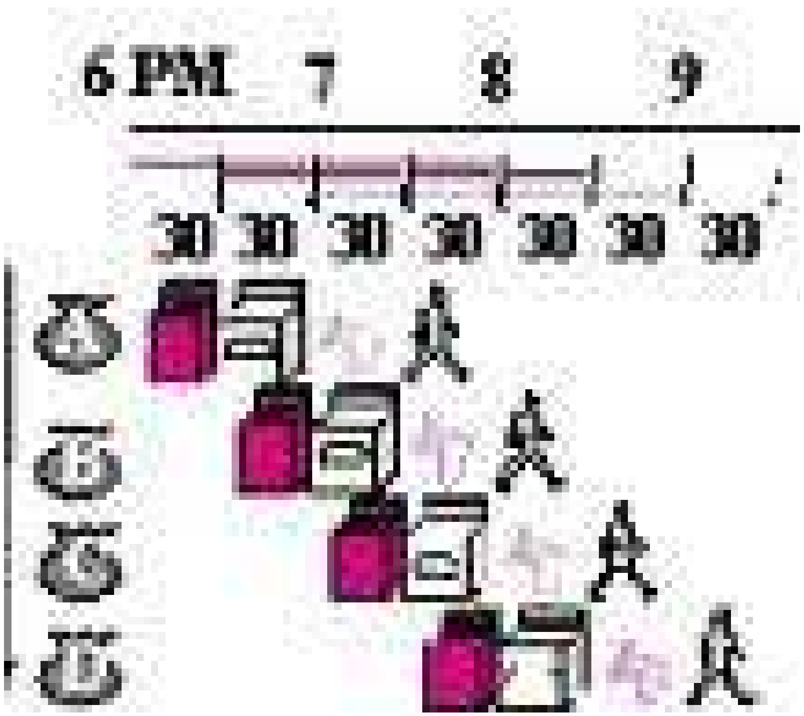
Latency vs. Throughput

Latency is the time from start to finish for a given task.

Throughput is the number of tasks completed in a given time period.

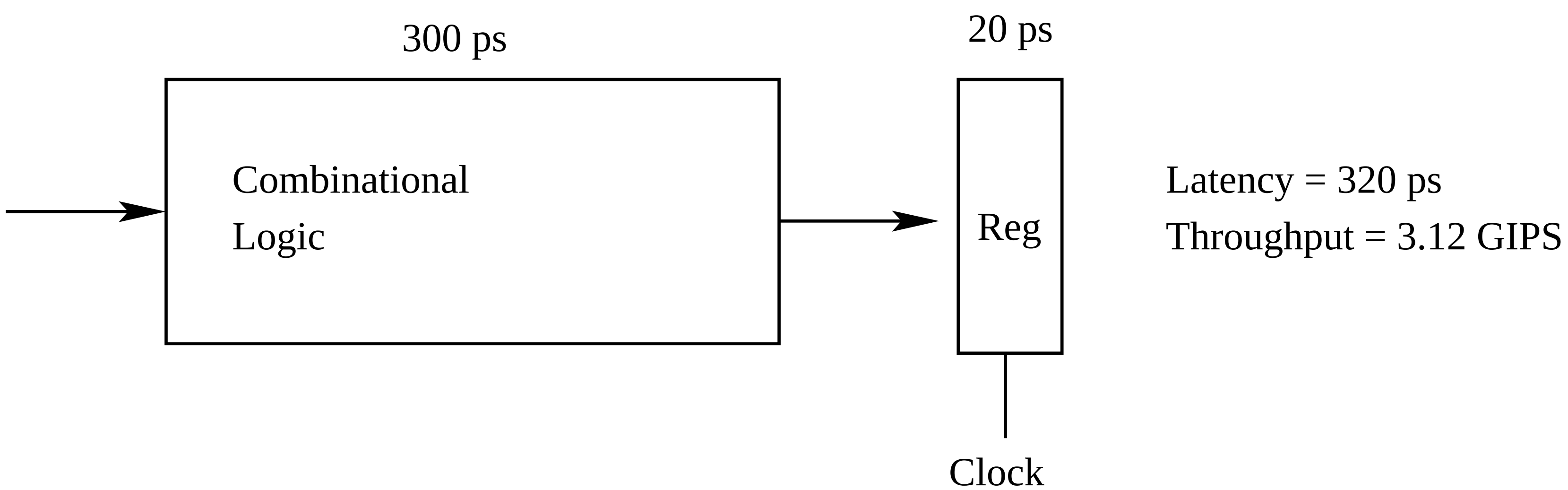
Example: suppose that each laundry stage (wash, dry, fold, stash) takes 30 minutes. But you have a laundromat with 4 washers, 4 driers, 4 folding stations, 4 stashing stations.

- What is the latency?
  Latency is 2 hours, because it still takes 2 hours to get any single load through the entire process.
- What is the highest possible throughput (per hour)?
  Throughput is (theoretically) 8 loads / hour since you can complete 8 loads every hour in steady state. How?



- Pipelining doesn't help *latency* of a single task; it helps *throughput* of the entire workload.
- Multiple tasks operate simultaneously using different resources.
- Potential speedup = number of stages.
- Unbalanced lengths of pipe stages may reduce speedup.
- Time to "fill" pipeline and time to "drain" it reduces speedup.
- May need to "stall" for dependencies.

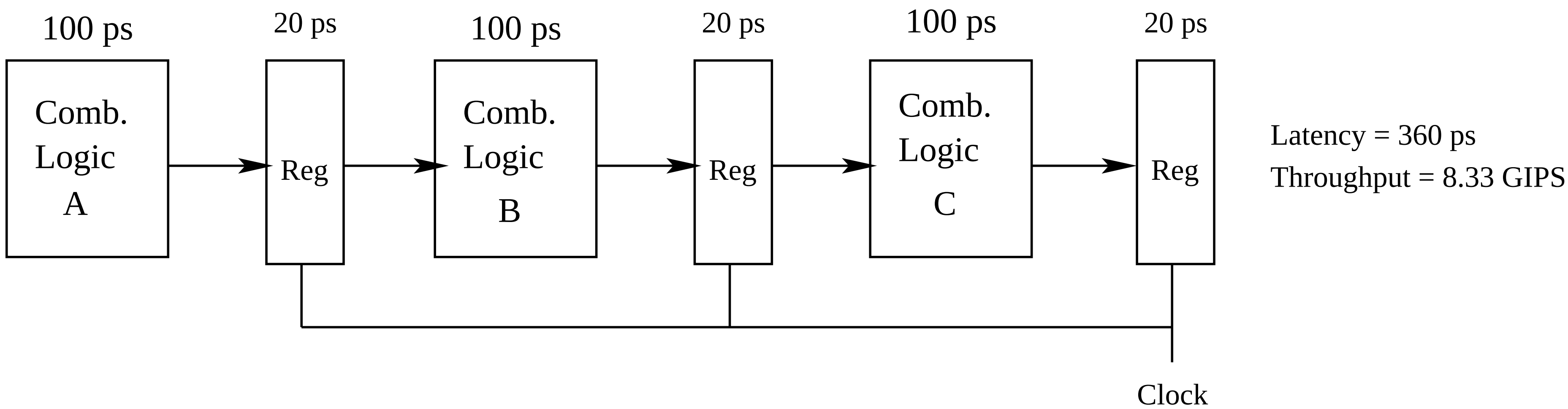
Computational Example



System

- Computation requires a total of 300 picoseconds.
- Needs an additional 20 picoseconds to save the result in the register.
- Must have a clock cycle of at least 320 ps. Why?

3-Way Pipelined Version

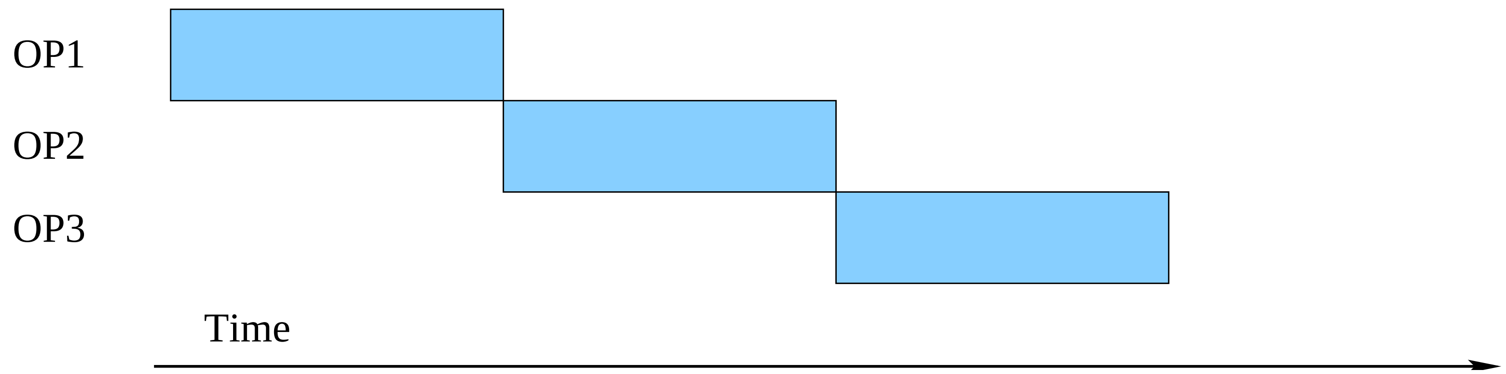


System

- Divide combinational logic into 3 blocks of 100 ps each.
- Can begin a new operation as soon as the previous one passes through stage A.
- Begin new operation every 120 ps. Why?
- Overall latency *increases*! It's now 360 ps from start to finish.

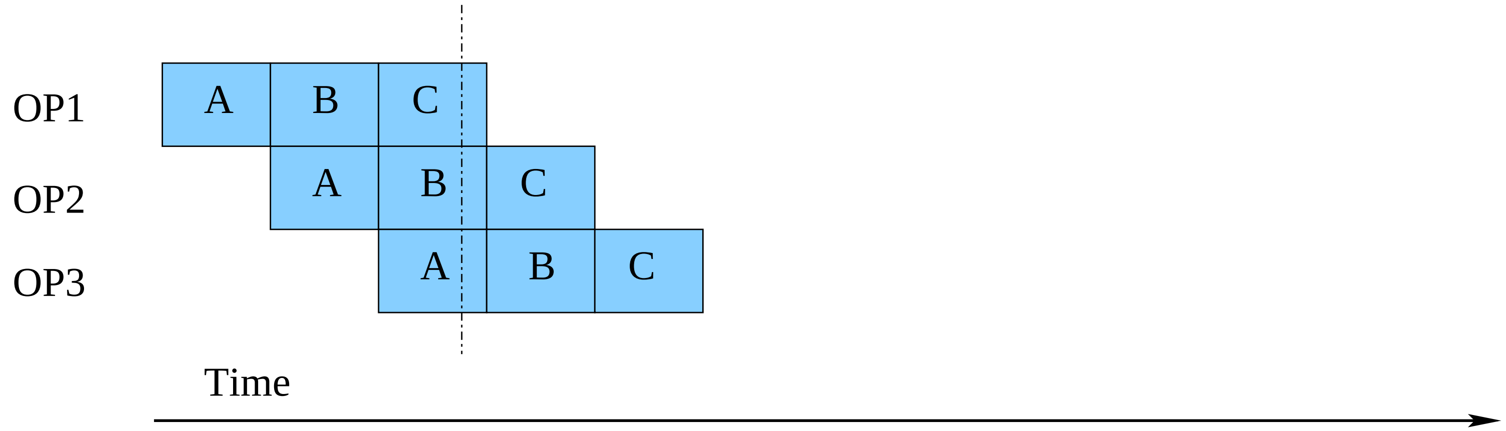
Pipeline Diagrams

Unpipelined



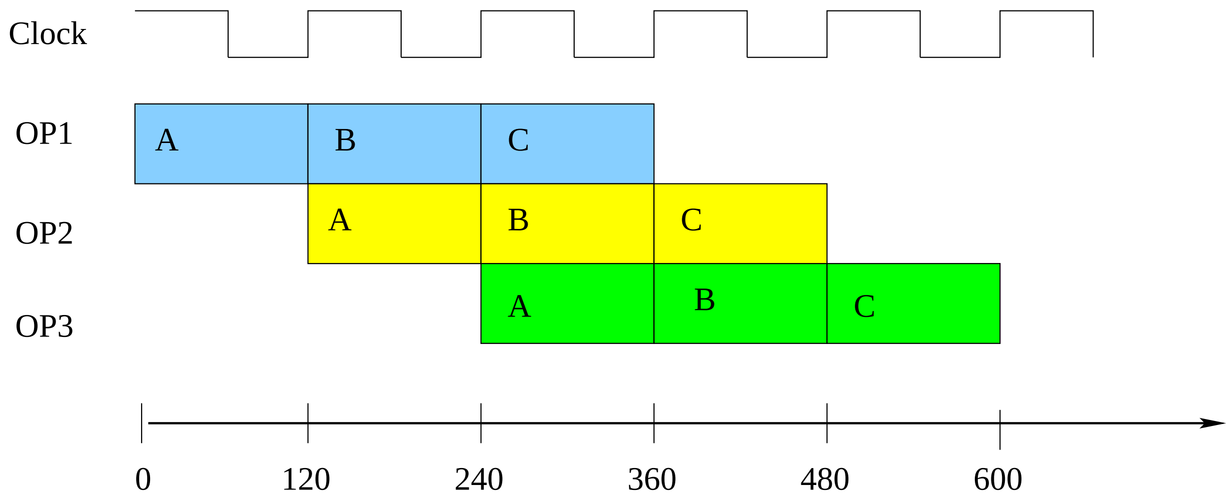
Cannot start new operation until the previous one completes.

3-Way Pipelined

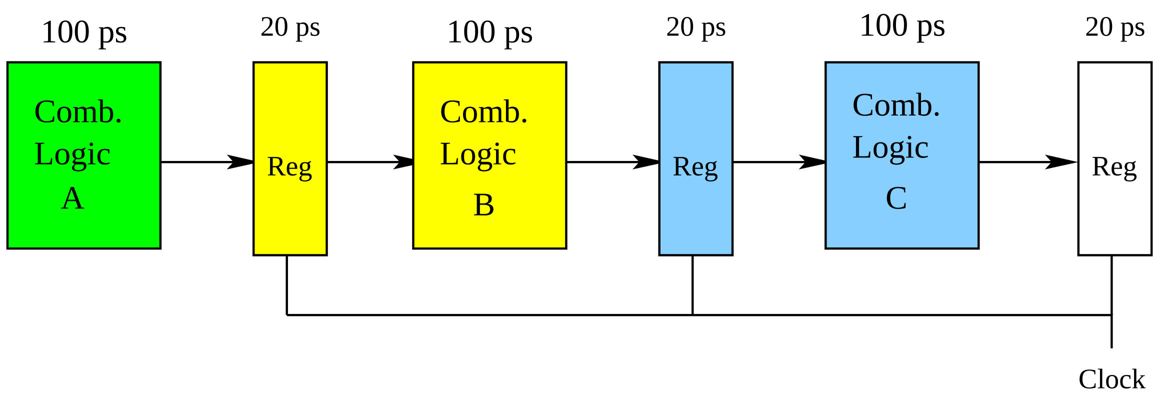


Up to 3 operations in process simultaneously.

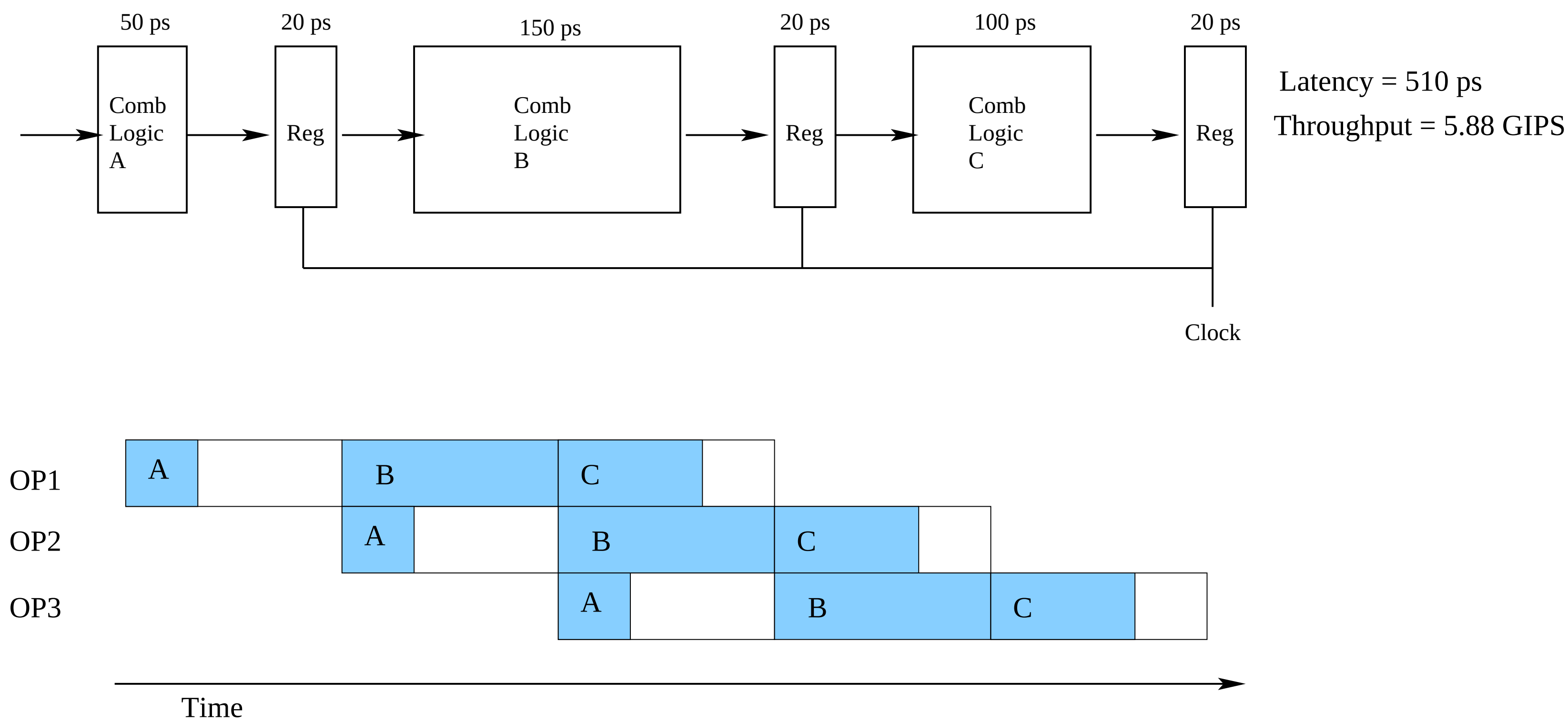
Operating a Pipeline



At time 300.

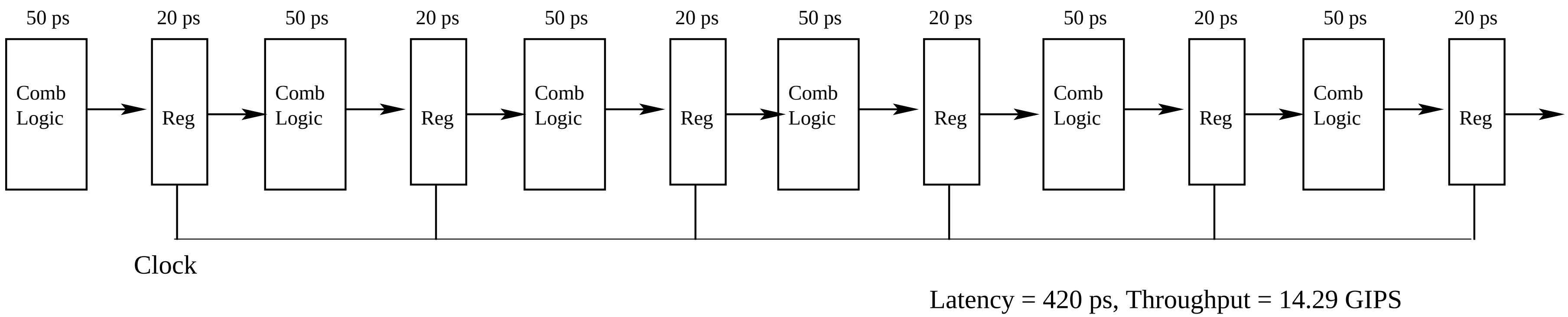


Limitations: Non-uniform Delays



- Throughput is limited by the slowest stage.
- Other stages may sit idle for much of the time.
- It's challenging to partition the system into balanced stages.

Limitations: Register Overhead



As you try to deepen the pipeline, the overhead of loading registers becomes more significant.

Percentage of clock cycle spent loading registers:

1-stage pipeline:	6.25%
3-stage pipeline:	16.67%
6-stage pipeline:	28.57%

High speeds of modern processor designs are obtained through very deep pipelining. (Some models of x86 have a pipeline of 20-24 stages.)

The Performance Equation

$$\text{CPU Time} = \frac{Seconds}{Program} = \frac{Instructions}{Program} * \frac{Cycles}{Instruction} * \frac{Seconds}{Cycle}$$

Clock Cycle Time

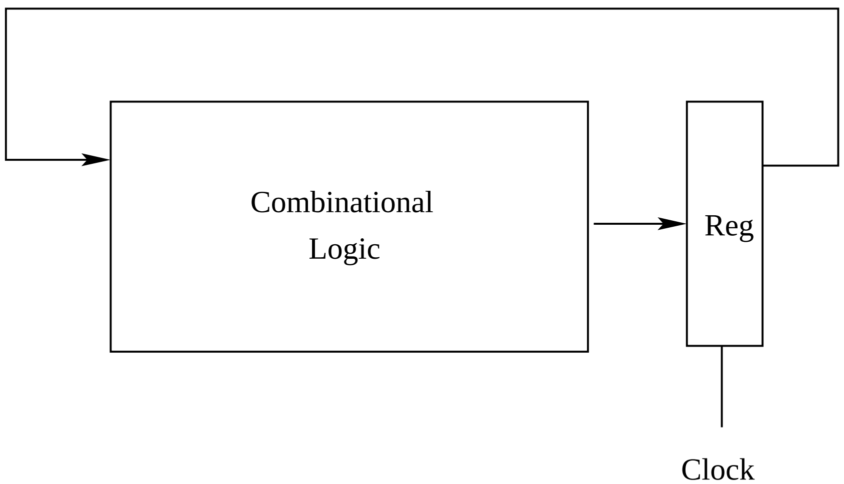
- Improves by a factor of almost N for N-deep pipeline.
- Not quite a factor of N due to pipeline overheads.

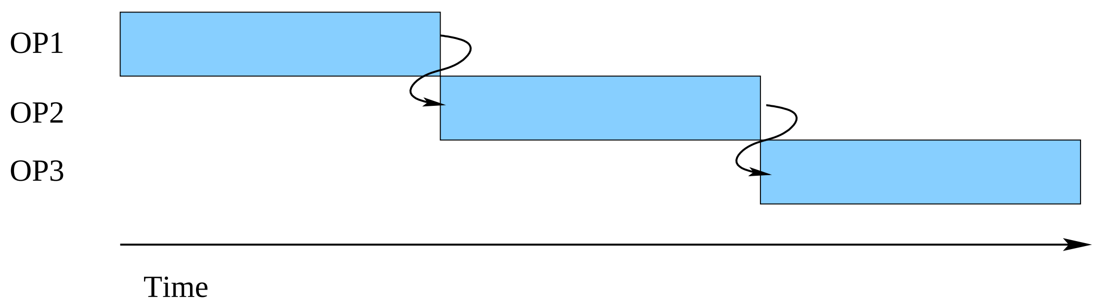
Cycles Per Instructions (CPI)

- In an ideal world, CPI would stay the same.
- An individual instruction takes N cycles.
- But we have N instructions in flight at a time.
- So, average $CPI_{pipe} = (CPI_{no-pipe} * N)/N$

Thus, performance can improve by up to a factor of N.

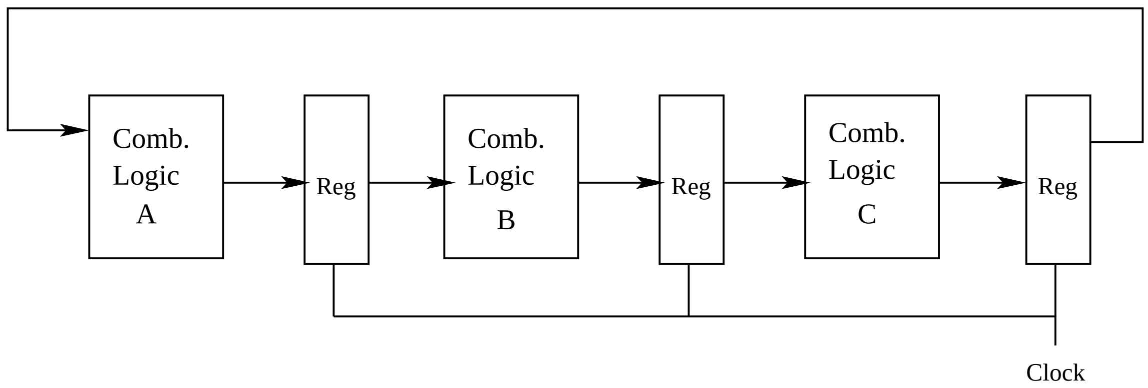
Data Dependencies

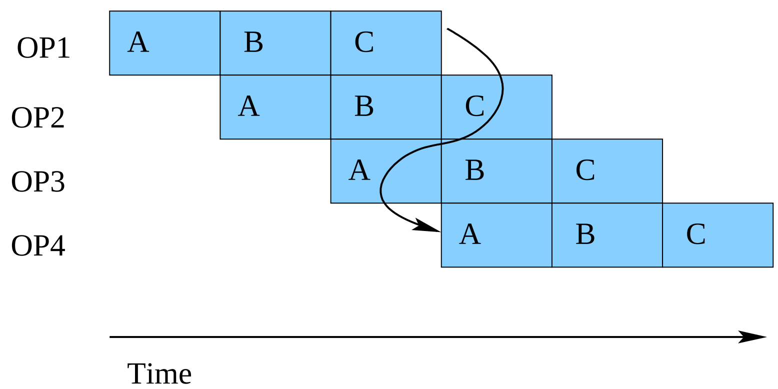




Sequential System: Each operation may depend on the previous one. (It doesn't matter for a sequential system. Why not?)

Data Hazards

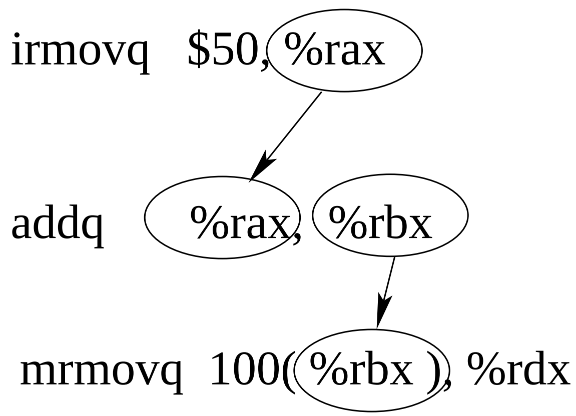




Pipelined System:

- Result does not feed back around in time for the next operation.
- Pipelining has changed the behavior of the system. *Alarm!!*

Data Hazards in Processors



Result from one instruction is used as an operand for another; called read-after-write (RAW) dependency.

This is very common in actual programs.

- Must make sure that our pipeline handles these properly and gets the right result.
- Should minimize performance impact as much as possible.

Control Hazards

A *control hazard* occurs if something interferes with the flow of control through the program. I.e., the PC is not determined quickly enough to allow fetching the next instruction.

```
        xorq    %rbx, %rbx
        je      Done
        irmovq  $100, %rax
        ret
Done:   irmovq  $200, %rax
        ret
```

When the `je` instruction moves from the fetch to decode stage, what is the next instruction to fetch? When will you know?

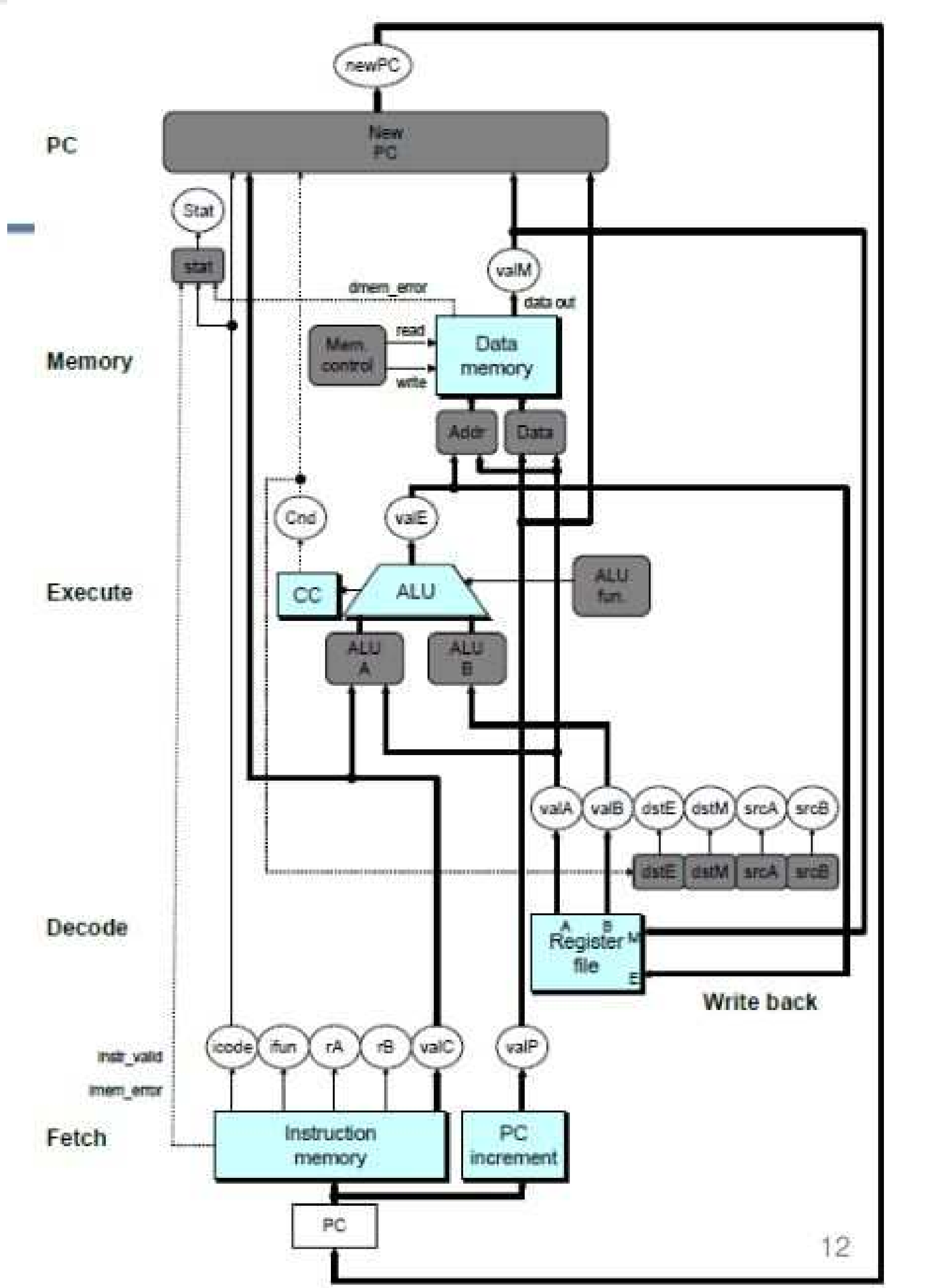
Pipeline Correctness

Pipeline Correctness Axiom: A pipeline is correct only if the resulting machine satisfies the ISA (nonpipelined) semantics.

That is, the pipeline implementation must deal correctly with potential data and control hazards. *Any program that runs correctly on the sequential machine must run on the pipelined version with the exact same results.*

SEQ Hardware

- Stages occur in sequence.
- One operation in process at at time.
- One stage for each logical pipeline operation.
  - **Fetch:** get next instruction from memory.
  - **Decode:** figure out what to do, and get values from regfile.
  - **Execute:** compute.
  - **Memory:** access data memory if needed.
  - **Write back:** write results to regfile, if needed.



SEQ+ Hardware

Still sequential implementation, but reorder PC stage to put at the beginning

PC Stage

- Task is to select PC for current instruction.
- Based on results computed by previous instruction.

Processor State

- PC is no longer stored in a register.
- But, can determine PC based on other stored information.